doi:10.1093/comjnl/bxh000

On the Usefulness of Fibonacci Compression Codes

Shmuel T. Klein, Miri Kopel Ben-Nissan

Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
Email: {tomi,kopel}@cs.biu.ac.il

Recent publications advocate the use of various variable length codes for which each codeword consists of an integral number of bytes in compression applications using large alphabets. This paper shows that another tradeoff with similar properties can be obtained by Fibonacci codes. These are fixed codeword sets, using binary representations of integers based on Fibonacci numbers of order $m \geq 2$. Fibonacci codes have been used before, and this paper extends previous work presenting several novel features. In particular, the compression efficiency is analyzed and compared to that of dense codes, and various table-driven decoding routines are suggested.

Received 00 Month 2004; revised 00 Month 2004

1. INTRODUCTION

In spite of the amazing advances in data storage technology, compression techniques are not becoming obsolete, and in fact research in data compression is flourishing as can be seen by the large number of papers published recently on the topic. In this work we concentrate on very large textual databases as those found in large Information Retrieval Systems. Such systems could contain hundreds of millions of words, which should be compressed by some method giving, in addition to good compression performance, also very fast decoding and the ability to search for the appearance of some strings directly in the compressed text. This paradigm of *compressed pattern matching* is a well established research topic that has generated a large literature in recent years, see [1, 2, 3, 4] to cite just a few.

Classical Huffman coding, when applied to individual characters, gives relatively poor compression, but when every word of a large textual database is considered as an atomic element to be encoded, this so-called Huffword variant may compete with the best other compression methods [5]. Yet the codewords of a binary Huffman code are not necessarily aligned on byte boundaries, which complicates both the decoding process and the ability to perform searches in the compressed file. The next step was therefore to pass to 256-ary Huffman coding, in which every codeword consists of an integral number of 8-bit bytes [3]. The loss incurred in the compression efficiency, which is only a few percent for large enough alphabets, is compensated for by the advantages of the easier processing.

When searches in the compressed text should also be supported, Huffman codes suffer from a problem of synchronization: denoting by $\mathcal{E}$ the encoding function, the compressed form $\mathcal{E}(x)$ of an element x may appear in the compressed text $\mathcal{E}(T)$, without corresponding to an actual occurrence of x in the text T, because the occurrence of $\mathcal{E}(x)$ is not necessarily aligned on codeword boundaries. This problem has been overcome in [6], relying on the tendency of Huffman codes to resynchronize quickly after errors, but the suggested solution is probabilistic and may produce wrong results. A deterministic solution has recently been given in [7]. As alternative, [3] propose to reserve the first bit of each byte as *tag*, which is used to identify the last byte of each codeword, thereby reducing the order of the Huffman tree to 128-ary. These *Tagged Huffman codes* have then been replaced by *End-Tagged Dense codes* (ETDC) in [8] and by (s, c)-*Dense codes* (SCDC) in [9]. The two last mentioned codes consist of fixed codewords which do not depend on the probabilities of the items to be encoded. Thus their construction is simpler than that of Huffman codes: all one has to do is to sort the items by non-increasing frequency and then assign the codewords accordingly, starting with the shortest ones.

SCDC is based on a pair of numbers (s, c), where the parameter s is chosen as $0 < s < 256$ and c is defined as $c = 256 - s$. Each codeword consists of a sequence of bytes $b_1 \cdots b_r$, the last of which satisfying $b_r < s$ and the others $b_i \geq s$ for $i < r$. ETDC is just the special case of SCDC for which $s = c = 128$.

ETDC and SCDC compress less than Huffman codes, but are better than Tagged Huffman codes as more codewords can be formed for every given length. In addition, coding is simplified and the good searching capabilities are maintained.

We show in this work that similar properties can be obtained by *Fibonacci codes*, which have been suggested in the context of compression codes for the unbounded transmission of strings [10] and because of their robustness against errors in data communication applications [11]. They are also studied as a simple alternative to Huffman codes in [12]. The main contribution of this paper is to show that Fibonacci codes are useful in this context and to present several new properties that have not been mentioned before, including (*i*) compression performance relative to other static methods like ETDC and SCDC, (*ii*) fast decoding techniques and (*iii*) support of compressed matching.

Note that Fibonacci codes have applications not only as alternatives to dense codes for large textual word-based compression systems. They are in particular mentioned in [13] as a good choice for compressing a set of small integers, and fast decoding as well as compressed searches may be important tools for such applications.

In the next section, we review the relevant features of Fibonacci codes of order $m \geq 2$. Section 3 then treats the new properties mentioned above and compares their performance with that of other static compression codes.

2. FIBONACCI CODES

Fibonacci numbers of order $m \geq 2$, denoted by $F_i^{(m)}$, are defined by the following recurrence relation:

$$F_n^{(m)} = F_{n-1}^{(m)} + F_{n-2}^{(m)} + \cdots + F_{n-m}^{(m)} \qquad \text{for } n > 0,$$

and the boundary conditions

$$F_0^{(m)} = 1 \qquad \text{and} \qquad F_n^{(m)} = 0 \quad \text{for } -m < n < 0.$$

For fixed order m, the number $F_n^{(m)}$ can be represented as a linear combination of the nth powers of the roots of the corresponding polynomial $P(m) = x^m - x^{m-1} - \cdots - x - 1$. $P(m)$ has only one real root that is larger than 1, which we shall denote by $\phi_{(m)}$, the other $m-1$ roots are complex numbers with norm < 1 (for $m = 2$, the second root is also real and its absolute value is < 1). Therefore, when representing $F_n^{(m)}$ as such a linear combination, the term with $\phi_{(m)}^n$ will be the dominant one, and the others will rapidly become negligible for increasing n.

For example, $m = 2$ corresponds to the classical Fibonacci sequence and $\phi_{(2)} = \frac{1+\sqrt{5}}{2} = 1.6180$ is the well-known golden ratio. As a matter of fact, the entire Fibonacci sequence can be obtained by $F_n^{(m)} = [a_{(m)}\phi_{(m)}^n]$, where $a_{(m)}$ is the coefficient of the dominating term in the above mentioned linear combination, and $[x]$ means that the value of the real number x is rounded to the closest integer, that is, $[x] = \lfloor x + 0.5 \rfloor$. Table 1 lists the first few elements of the Fibonacci sequences of order up to 6. The column headed *General Term* brings the values of $a_{(m)}$ and $\phi_{(m)}$. For larger n, the numbers $a_{(m)}\phi_{(m)}^n$ are usually quite close to integers, e.g., $a_{(2)}\phi_{(2)}^{10} = 88.998$ and $a_{(3)}\phi_{(3)}^{13} = 1705.01$.

The standard representation of an integer as a binary string is based on a numeration system whose basis elements are the powers of 2. If B is represented by the k-bit string $b_{k-1}b_{k-2}\cdots b_1b_0$, then $B = \sum_{i=0}^{k-1} b_i 2^i$. But many other possible binary representations do exist, and those using the Fibonacci sequences as basis elements have some interesting properties. Let us first consider the standard Fibonacci numbers of order 2.

Any integer B can be represented by a binary string of length r, $c_r c_{r-1}\cdots c_2 c_1$, such that $B = \sum_{i=1}^{r} c_i F_i^{(2)}$. The representation will be unique if one uses the following procedure to produce it: given the integer B, find the largest Fibonacci number $F_r^{(2)}$ smaller or equal to B; then continue recursively with $B - F_r^{(2)}$. For example, $45 = 34 + 8 + 3$, so its binary Fibonacci representation would be 10010100. As a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s.

This property can be exploited to devise an infinite code whose set of codewords consists of the Fibonacci representations of the integers: to assure the code being uniquely decipherable (UD), each codeword is prefixed by a single 1-bit, which acts like a *comma* and permits to identify the boundaries between the codewords. The first few elements of this code would thus be $\{u_1, u_2, \ldots\}$ = {**11**, **1**10, **1**100, **1**101, **1**1000, **1**1001,...}, where the separating 1 is put in boldface for visibility. A typical compressed text could be **1**100**11**00**11**01**11**1101, which is easily parsed as $u_6u_3u_4u_1u_4$. Though being UD, this is not a prefix code, so decoding may be somewhat more involved. In particular, the first codeword 11, which is the only one containing no zeros, complicates the decoding, because if a run of several such codewords appears, the correct decoding of the codeword preceding the run depends on the parity of the length of the run. Consider for example the encoded string 11011111110: a first attempt to parse it as 110 | 11 | 11 | 11 | 10 = $u_2u_1u_1u_1$10 would fail, because the tail 10 is not a codeword; hence only when trying to decode the fifth codeword do we realize that the first one is not correct, and that the parsing should rather be 1101 | 11 | 11 | 110 = $u_4u_1u_1u_2$.

To overcome this problem, [10, 11] suggest to reverse all the codewords, yielding the set $\{v_1, v_2, \ldots\}$ = {11, 011, 0011, 1011, 00011, 10011,...}, which is a prefix

$F_n^{(m)}$	General Term	1	2	3	4	5	6	7	8	9	10	11	12	13
$m=2$	$0.7236\ (1.6180)^n$	1	2	3	5	8	13	21	34	55	89	144	233	377
$m=3$	$0.6184\ (1.8393)^n$	1	2	4	7	13	24	44	81	149	274	504	927	1705
$m=4$	$0.5663\ (1.9275)^n$	1	2	4	8	15	29	56	108	208	401	773	1490	2872
$m=5$	$0.5379\ (1.9659)^n$	1	2	4	8	16	31	61	120	236	464	912	1793	3525
$m=6$	$0.5218\ (1.9836)^n$	1	2	4	8	16	32	63	125	248	492	976	1936	3840

TABLE 1: *Fibonacci numbers of order* $m = 2, 3, 4, 5, 6$

code, since all codewords are terminated by 11 and this substring does not appear anywhere in any codeword, except at its suffix. In addition, we show below that having a reversed representation, with the bits corresponding to increasing basis elements running from left to right rather than as usual, is advantageous for fast decoding. Table 2 brings a larger sample of this set of codewords in the column headed Fib2. Note that the order of the elements is not lexicographic, e.g., 10011 precedes 01011.

The generalization to higher order seems at first sight straightforward: any integer B can be uniquely represented by the string $d_s d_{s-1} \cdots d_2 d_1$ such that $B = \sum_{i=1}^{s} d_i F_i^{(m)}$ using the iterative encoding procedure mentioned above. In this representation, there are no consecutive substrings of m 1s. For example, the representations of the integers 10, 11, 12 and 13 using $F^{(3)}$ are, respectively, 1011, 1100, 1101 and 10000. But simply adding now $m-1$ 1's as commas and reversing the strings does not yield a prefix code for $m > 2$, and in fact the code so obtained is not even UD. For example, for $m = 3$, the above numbers would give the codewords $\{v_{10}, \ldots, v_{13}\} = \{1101\mathbf{11}, 0011\mathbf{11}, 1011\mathbf{11}, 0000\mathbf{11}\}$, but the encoding of the fourth element of the sequence would be $v_4 = 001\mathbf{11}$, which is a prefix of v_{11}. The string 0011110111 could be parsed both as $00111 \,|\, 10111 = v_4 v_5$ and as $001111 \,|\, 0111 = v_{11} v_2$.

The problem stems from the fact that for $m > 2$, there can be more than one leading 1 in the representation of an integer, so adding $m-1$ 1s may give a string of up to $2m-2$ consecutive 1s. The fact that a string of m 1s appears only as a suffix is thus only true for $m = 2$. To turn the sequence into a prefix code, the definition has to be amended as follows: the set Fib$m$ will be defined as the set of binary codewords of lengths $\geq m$, such that every codeword contains exactly one occurrence of the substring consisting of m consecutive 1s, and this occurrence is the suffix of every codeword. The first elements of these codes for $m \leq 4$ are given in Table 2. For $m = 2$, this last definition is equivalent to the one above based on the representation with basis elements $F_n^{(2)}$; for $m > 2$, only a subset of the corresponding codewords is taken. There is nevertheless a connection between the codewords and the higher order Fibonacci numbers: for $m \geq 2$, and

index	Fib2	Fib3	Fib4
1	11	111	1111
2	011	0111	01111
3	0011	00111	001111
4	1011	10111	101111
5	00011	000111	0001111
6	10011	100111	1001111
7	01011	010111	0101111
8	000011	110111	1101111
9	100011	0000111	00001111
10	010011	1000111	10001111
11	001011	0100111	01001111
12	101011	1100111	11001111
13	0000011	0010111	00101111
14	1000011	1010111	10101111
15	0100011	0110111	01101111
16	0010011	00000111	11101111
17	1010011	10000111	000001111
18	0001011	01000111	100001111
19	1001011	11000111	010001111
20	0101011	00100111	110001111
21	00000011	10100111	001001111
22	10000011	01100111	101001111
23	01000011	00010111	011001111
24	00100011	10010111	111001111
25	10100011	01010111	000101111
26	00010011	11010111	100101111
27	10010011	00110111	010101111
28	01010011	10110111	110101111
29	00001011	000000111	001101111
30	10001011	100000111	101101111
31	01001011	010000111	011101111
32	00101011	110000111	0000001111
33	10101011	001000111	1000001111
34	000000011	101000111	0100001111
35	100000011	011000111	1100001111

TABLE 2: *Fibonacci codes of order* $m = 2, 3, 4$

$n \geq 0$, the code Fib$m$ consists of

$$F_n^{(m)} \quad \text{codewords of length} \quad n+m. \tag{1}$$

This is visualized in Table 2, where for each code, blocks of codewords of the same length are separated by horizontal lines. Within each such block of lengths $\geq m+2$ for Fib$m$, the prefixes of the codewords obtained by removing the terminating string of 1s correspond to consecutive integers in the representation based on $F^{(m)}$. For decoding, the Fibonacci representation will thus be used to get the relative index within the block, to which the starting index of the given block has to be added. This is identical to the decoding procedure of *canonical Huffman codes*, for which each block of codewords of a given length consists of the standard binary representation of consecutive integers, see [14].

3. COMPRESSION BY FIBONACCI CODES

We now turn to an investigation of some properties of the Fibonacci codes and in particular compare them with the dense codes ETDC and SCDC. The

latter were introduced in [9, 8] as alternatives to Tagged Huffman codes, improving these because of the following advantages:

(i) better *compression* ratios;
(ii) simpler *vocabulary* representation;
(iii) simpler and faster *decoding*;
(iv) same *searching* possibilities.

We show that on all these criteria, Fibonacci codes are a plausible alternative to ETDC and SCDC, improving on the first, being equivalent on the second and inferior on the last two. A simple implementation of decoding and searching could be as much as 100 times slower than for the dense codes, and we show how to accelerate both procedures, reducing the advantage of SCDC to 2 to 3-fold. We also add *robustness* as fifth criterion, for which the Fibonacci codes are preferable. Our conclusion is that overall, Fibonacci codes may be an attractive substitute for dense codes in certain applications.

A comprehensive study of the Fibonacci codes should possibly also include comparisons with other alternatives, like standard binary Huffman codes or arithmetic coding. We shall, however, keep the focus in this paper on the comparison against the dense codes ETDC and SCDC, and mention the others only briefly, first, because such comparisons can already be found in the papers on dense codes, and second, because the choice of an encoding scheme may be considered as a package deal: if one had only a single application in mind, then different codes could be chosen according to the intended application. So if the only criterion is compression, one would use arithmetic coding, which practically achieves the entropy bound, and if speed is the only concern, one could use ETDC for example. But the assumption here is, like in [8] and [9], that one chooses a single code which should be useful for various applications and on several criteria.

Classical 2-ary Huffman coding gives very good compression when applied on the different words of a large text, and the excess of the entropy, the information theoretic lower bound, is below 0.3% on our examples, see Table 3 below. Simple bit-wise decoding is slow, but several table driven decoding methods have been devised, achieving a significant speedup [15]. Searching in Huffman encoded texts has to deal with synchronization problems, but can be done, as mentioned in the introduction.

Arithmetic coding gives optimal compression, and examples of typical compression ratios are included in Table 3 in the column headed *Entropy*, though the savings relative to Huffman codes are generally very low [16]. But the processing for both encoding and decoding is much slower than for all the alternatives mentioned in this paper, and searching within the compressed text is impossible, since it is not true that different occurrences of a word in the text are always encoded by the same bitstring.

3.1. Compression efficiency

Let us first compare the number of codewords of each of the codes for a given length. As a rough approximation, ETDC utilize 7 of the 8 bits of each byte, so that the number of codewords grows like $2^{7k/8} = 1.834^k$, where k is the number of bits. This should be compared with the Fibonacci codes, for which the number grows roughly like $\phi_{(m)}^k$. Thus we expect the codes based on standard Fibonacci numbers to have less codewords than ETDC, but for $m > 2$, the Fib$m$ codes are denser than ETDC.

More precisely, the number of codewords of length up to M bytes for ETDC is $128 \sum_{i=1}^{M} 128^{i-1} \simeq 128^{M-1} = 2^{7M}$. For Fib$m$, the last m bits of each codeword are reserved, so using eq. (1), the number of codewords with up to $8M$ bits is approximately

$$\sum_{i=1}^{8M-m} F_i^{(m)} \simeq a_{(m)} \sum_{i=1}^{8M-m} \phi_{(m)}^i \simeq \frac{a_{(m)}}{\phi_{(m)} - 1} \phi_{(m)}^{8M-m+1},$$

which should be compared with 2^{7M}. We get that the number of codewords for Fib$m$ will be larger if

$$\left(\frac{\phi_{(m)}^8}{2^7} \right)^M > \frac{(\phi_{(m)} - 1)\phi_{(m)}^{m-1}}{a_{(m)}}. \tag{2}$$

But for $m = 2$, the constant $\phi_{(2)}^8/2^7 = 0.367$ is smaller than 1, while the right hand side of (2) is $1.38 > 1$, so that for any length M, ETDC has more codewords than Fib2. For $m \geq 3$, we get from (2) that

$$M > \frac{(m-1)\log_2 \phi_{(m)} + \log_2(\phi_{(m)} - 1) - \log_2 a_{(m)}}{8 \log_2 \phi_{(m)} - 7}, \tag{3}$$

so that asymptotically all the Fib$m$ codes are denser, and their number of codewords is larger than that of ETDC if M is at least 67, 7, 6 and 7, for $m = 3, 4, 5, 6$, respectively. These values derive from the approximate lower bound of equation (3) and coincide with the values obtained using precise integer computations. Of course, the fact that Fib$m$ codes are denser in the limit has no impact on real life distributions, as $M = 4$ suffices already for huge alphabets.

However, the codes should not be compared only on the basis of the number of their elements. This would correspond to a uniform distribution, for which it makes no sense to use variable length codes anyway. For non-uniform distributions, the advantage of the Fibonacci codes should be even more evident, as because of the possibility of using a larger set of different lengths, the codes can approach the optimal codeword lengths more closely, and in particular assign codewords that are shorter than 1 byte to the elements of highest frequency. The rigidity of ETDC, which assigns 1 byte codewords only to the 128 most frequent elements, and then already passes to codewords of length 2 bytes was the main motivation for the development of the (s, c)-codes; Fibonacci codes have also codewords shorter than one byte.

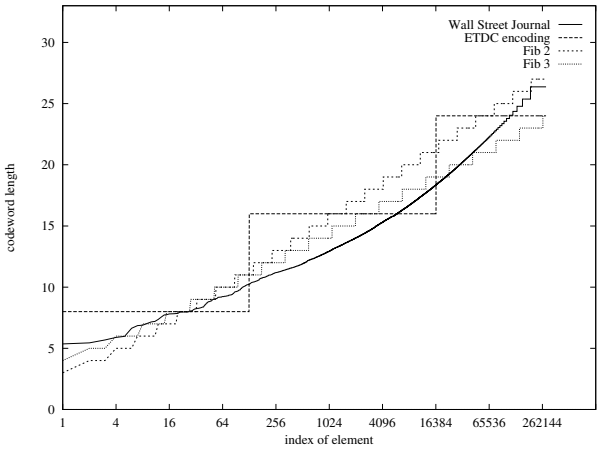

FIGURE 1: *Codeword lengths*

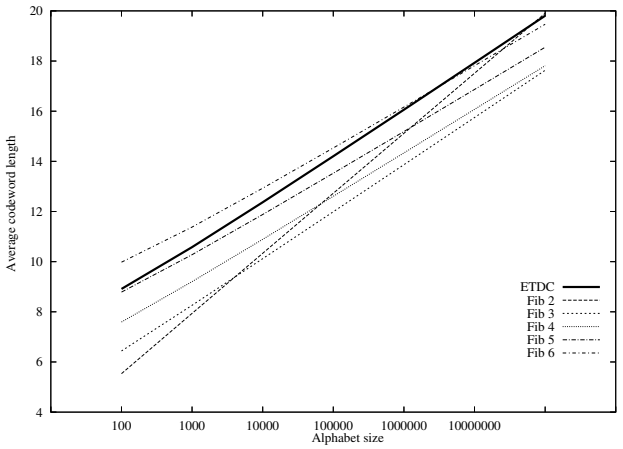

FIGURE 2: *Zipf averages*

Figure 1 plots the lengths of the codewords as a function of the index of the element on a logarithmic scale for ETDC, Fib2 and Fib3. In addition, there is also a (seemingly continuous) curve corresponding to a real life distribution, 500 MB (87 million words) of the *Wall Street Journal* [17]; the plot corresponds to an ideal encoding of the different words, using $-\log_2 p_i$ bits to encode an element with probability p_i, as could be approximated by an arithmetic encoder. We see how the Fibonacci curves approach the real distribution by a series of small steps much closer than the larger steps of ETDC.

To compare the codes analytically on more realistic distributions than the uniform one, consider Zipf's law [18], which is believed to govern the distribution of the words in a large natural language text. It is defined by the weights $p_i = 1/(i\,H_N)$, for $1 \le i \le N$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number[1]. It is well known that $H_n \simeq \ln n - \gamma$, where $\gamma = 0.5772$ is Euler's constant. Since Fib$m$ and ETDC are step functions, it is convenient to have a notation for the indices of the last elements of every codeword length. Let $L_E(k)$ and $L_m(k)$ denote, respectively, these indices for ETDC and Fib$m$, $m \ge 2$. Thus $L_E(8) = 128$, $L_E(16) = 16512$, $L_E(24) = 2113664$; $L_2(k) = 1, 2, 4, 7, 12, 20, 33, \ldots$, $L_3(k) = 1, 2, 4, 8, 15, 28, \ldots$ and $L_4(k) = 1, 2, 4, 8, 16, 31, \ldots$, for $k \ge m$, as can be seen in Table 2. Similarly, let $M_E(N)$ and $M_m(N)$ denote the number of different codeword lengths, which is the number of blocks of codewords of equal length for an alphabet of size N. Since approximately $(128)^{M_E(N)} = N$ and $a_{(m)}\phi_{(m)}^{M_m(N)}/(\phi_{(m)} - 1) = N$, we get that

$$M_E(N) \simeq \frac{\ln N}{\ln 128}$$

and

$$M_m(N) \simeq \frac{\ln(\phi_{(m)} - 1) + \ln N - \ln a_{(m)}}{\ln \phi_{(m)}}.$$

Denoting the ith codeword by C_i, the average length of a codeword using a Zipf distribution is then

$$\sum_{i=1}^{N} \frac{1}{iH_N}|C_i| \simeq \frac{1}{H_N}\sum_{k=1}^{M_E(N)} 8k \sum_{L_E(k-1)+1}^{L_E(k)} \frac{1}{i} \quad \text{for ETDC}$$
$$\simeq \frac{1}{H_N}\sum_{k=1}^{M_m(N)} (k+m-1) \sum_{L_m(k-1)+1}^{L_m(k)} \frac{1}{i} \quad \text{for Fib}m.$$

But

$$\begin{aligned}\sum_{L(k-1)+1}^{L(k)} \frac{1}{i} &= H_{L(k)} - H_{L(k-1)} \\ &\simeq \ln L(k) - \gamma - (\ln L(k-1) - \gamma) \\ &= \ln \frac{L(k)}{L(k-1)},\end{aligned}$$

and the ratio of consecutive L values is roughly 128 for ETDC and $\phi_{(m)}$ for Fib$m$. We thus get that the average is

$$\frac{4\ln 128}{\ln N - \gamma} M_E(N)(M_E(N)+1) \qquad \text{for ETDC,}$$

and for Fib$m$, it is, after simplification,

$$\frac{\ln \phi_{(m)}}{2(\ln N - \gamma)}\; M_m(N)\; \Big(M_m(N) + 2m - 1\Big).$$

Figure 2 plots these average values for N up to 10^8 for ETDC (the bold line) and Fib$m$, with $2 \le m \le 6$. As all the functions are roughly proportional to $\log N$, we get almost straight lines on the displayed logarithmic scale. We see that ETDC has a consistently longer average than the Fibonacci codes, at least for $m \le 5$ and up to alphabet sizes of 10^8. The additional overhead may

[1] Actually, a more precise definition of the law is $p_i = 1/i^\theta\, H_N(\theta)$ for some constant $\theta > 1$, where $H_n(\theta) = \sum_{j=1}^{n}(1/j^\theta)$, but we shall stick to the simpler definition; according to Wikipedia, θ is just slightly larger than 1.

Language	size	Entropy	excess in % over the entropy lower bound								
	words	bits	Huffman	Fib2	Fib3	Fib4	Fib5	Fib6	ETDC	(s,c)	
English	289,101	11.173	0.26	10.63	5.35	10.72	18.51	27.01	16.09	(189,67)	14.12
French	439,191	10.442	0.29	8.31	4.77	11.09	19.61	28.71	14.82	(185,71)	13.10
Hebrew	296,933	13.003	0.23	15.35	6.82	10.44	16.74	23.82	15.20	(176,80)	13.90
Artificial	1,000,000	13.378	0.20	12.42	3.68	7.04	13.02	19.90	14.07	(171,85)	13.32

TABLE 3: *Average codeword lengths and excesses over the entropy*

index		Λ		00011		01		100010	
dec	bin	S	R	S	R	S	R	S	R
⋮	⋮								
220	11011100	C_8	00	C_5C_4	00	C_2C_2	00	C_{317}	00
221	11011101	C_8	01	C_5C_4	01	C_2C_2	01	C_{317}	01
222	11011110	C_8	10	C_5C_4	10	C_2C_2	10	C_{317}	10
223	11011111	C_8	11	C_5C_4	11	C_2C_2	11	C_{317}	11
224	11100000	C_1	00000	C_5	1100000	C_2	100000	C_{43}	00000
225	11100001	C_1	00001	C_5	1100001	C_2	100001	C_{43}	00001
226	11100010	C_1	00010	C_5	1100010	C_2	100010	C_{43}	00010
227	11100011	C_1	00011	C_5	1100011	C_2	100011	C_{43}	00011
⋮	⋮								

TABLE 4: *Partial decoding tables*

reach 33% for alphabets as small as 1000 elements, but even for $N = 1000000$, the overhead of ETDC relative to Fib2, Fib3 and Fib4 is 1, 10 and 7%, respectively.

For a comparison of real data and different languages, the following sets were used, beside *Wall Street Journal*: the data for French was collected from the *Trésor de la Langue Française*, a database of 680 MB of French language texts (115 million words) of the 17^{th}–20^{th} centuries [19], and for Hebrew, the data was a part of the *Responsa Retrieval Project*, 100MB of Hebrew and Aramaic texts (15 million words) written over the past ten centuries [20]. All these texts have been encoded as sequences of words. Table 3 lists the sizes of the alphabets (number of words) and the entropy of each distribution in bits per codeword, which gives a lower bound on the possible average codeword lengths, as could be approached by an arithmetic encoder. Then follow the values for 2-ary Huffman, Fib$m$, ETDC and SCDC codes, all given as the excess, in percent, over the entropy, the information theoretic lower bound. For SCDC, the best (s,c) pair was chosen for each of the distributions, and this optimal pair appears also in the last column. The table also gives values for an artificial Zipf distribution of size 10^6.

Among the static codes considered here, the best results are consistently given by Fib3, saving additional 6–9% over ETDC or the best SCDC, and exceeding the entropy or a Huffman code only by 4–7%. Even Fib2 encoding improves by 1–4% on the English, French and artificial texts, and only for Hebrew, Fib2 is marginally inferior (by 1.2%) to the best SCDC.

3.2. Vocabulary representation

There are many different Huffman codes for a given probability distribution. Even if canonical Huffman codes are used, one still needs to store the number of codewords of every length. The size of this additional storage is of course negligible relatively to the sizes of the other files involved, but the fact that the set of codewords will vary according to the distribution at hand puts an additional burden on both encoder and decoder. On the other hand, ETDC, SCDC and the Fib$m$ codes are fixed sets, not depending on the frequencies. The codewords are thus stored once and for all, and from that point of view, ETDC, SCDC and the Fib$m$ codes are equivalent.

3.3. Fast decoding

A major reason for abandoning the optimal binary Huffman codes and using instead 256-ary Huffman codes or SCDC is to obtain faster decoding, since each codeword consists of an integral number of bytes. Though fast methods for the decoding of general 2-ary Huffman codes have been devised [15, 14, 24], they cannot be faster than the decoding of byte aligned Huffman codes, which do not need time consuming bit manipulations and can perform the whole process at

the byte level. On the other hand, the decoding of byte aligned Huffman codes has extensively been compared with that of ETDC and SCDC in [21], and found to be roughly equivalent: Huffman decoding is reported to be up to 3% faster.

The Fib$m$ codes are also of variable lengths but not necessarily byte-aligned, so the naive approach to decoding, involving many bit manipulations, will be more costly than for the alternatives. This section deals with ways to accelerate the decoding process. As mentioned above, this is important for any application using Fibonacci codes, not only as alternatives to dense codes, and we show a significant improvement over the standard decoding procedure. In comparison with ETDC and SCDC, decoding is still slower, but the inferiority of the Fib$m$ codes is greatly reduced.

As a decoding baseline, consider the simple bitwise decoding procedure as follows: the Fib$m$ encoded text is denoted $\mathsf{T}_1\mathsf{T}_2\mathsf{T}_3\cdots$, and an array $\mathsf{Start}_m[j]$ gives the index of the first element of length j in the Fib$m$ code. Thus $\mathsf{Start}_2[] = 1, 2, 3, 5, 8, \ldots$, $\mathsf{Start}_3[] = 1, 2, 3, 5, 9, 16, \ldots$, $\mathsf{Start}_4[] = 1, 2, 3, 5, 9, 17, 32, \ldots$, as can be seen in Table 2. Note also that using the notation of the previous section, where the index of the *last* element for each length was needed, we have $\mathsf{Start}_m[j+1] = L_m(j) + 1$ for $j \geq m$. Let $\mathsf{Word}[j]$ be the element encoded by the j-th codeword; i will point to the bit preceding the first bit of a codeword, ℓ is the number of bits decoded so far within the current codeword, and rel_ind is the relative index of the current codeword within the list of those of length ℓ; $F^{(m)}[j]$ is the j-th Fibonacci number of order m, as in Table 1. The formal naive decoding is then given by:

```
i ⟵ 0   ℓ ⟵ 0
rel_ind ⟵ 0
while i < length of encoded text in bits
    if T_{i+ℓ+1} ··· T_{i+ℓ+m} = 11 ··· 1
        /* m consecutive 1s */
        output ⟵ Word[ Start_m[ℓ + m] + rel_ind ]
        i ⟵ i + ℓ + m   ℓ ⟵ 0
        rel_ind ⟵ 0
    else
        ℓ ⟵ ℓ + 1
        rel_ind ⟵ rel_ind + T_{i+ℓ} × F^(m)[ℓ]
```

In words, the input is processed bit by bit. If the last m bits are a string of 1s, a codeword has been detected and can be sent to output, otherwise the relative index of the current codeword is incrementally calculated by adding the corresponding Fibonacci numbers.

3.3.1. Partial decoding tables

To improve the bitwise approach, we adapt, in a first attempt, a method originally suggested for the fast decoding of Huffman codes, which suffer from the same problem as the variable length Fib$m$ codes. The method uses a set of *partial decoding tables* that are prepared in advance and depend only on the code, not on the actual text to be decoded, by means of which the decoding is then performed by blocks of k bits at a time, rather than bit per bit. Typically, $k = 8$ or 16. One therefore gets a time/space tradeoff, where faster decoding comes at the cost of storing larger tables. The range of the parameter k is nevertheless restricted, because too large tables, even if they fit into the available RAM, may cause more page faults and cache misses, which may cancel the speed advantages. The method has first been presented in [22] and has been reinvented several times, e.g., [23].

The basic scheme is as follows. The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (S, R), where S is a sequence of characters and R is the label of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of the first table contains the longest possible sequence S of characters that can be decoded from the k-bit block representing the integer i (S may be empty when there are codewords of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P of more than one codeword; there will be one table for each of the possible prefixes P, in particular, if $P = \Lambda$, where Λ denotes the empty string, the corresponding table is the first one.

The table for $P \neq \Lambda$ is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P to the binary representation of i. More formally, in the table corresponding to P, the i-th entry, $\mathsf{Tab}[i, P]$, is defined as follows: let B be the bit-string composed of the juxtaposition of P to the left of the k-bit binary representation of i. Let S be the (possibly empty) longest sequence of characters that can be decoded from B, and R the remaining undecipherable bits of B; then $\mathsf{Tab}[i, P] = (S, R)$. Table 4 brings a part of these partial decoding tables for $m = 3$ and $k = 8$, showing certain lines for selected values of P. The first columns are the indices to be decoded in both decimal and binary notations; the decoding tables are labeled by the corresponding prefixes P.

The general decoding routine is thus extremely simple: denoting the ith byte of the encoded text by $\mathsf{Text}[i]$, one performs

```
R ⟵ Λ
for i ⟵ 1 to length of text in bytes
    (output, R) ⟵ Tab[Text[i], R]
```

As example, consider the input string displayed in Figure 3. The different codewords appear in alternating grey tones, the decimal value of the bytes is displayed above and the output generated by the decoding procedure appears below. For the given example, the bit string in the first byte is the binary representation

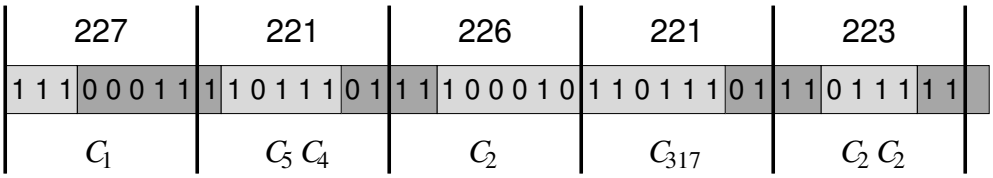

FIGURE 3: *Example of decoding by partial decoding tables*

of 227, so one accesses the first table at entry 227, yielding as output C_1 and a remainder suffix 00011. The next table accessed is thus that labeled 00011, at entry 221, which is the value of the next byte; this table contains at entry 221 the partial decoding of the string 0001111011101, which yields the characters C_5C_4 and the remainder $P = 01$, etc. Refer to Table 4, where the entries used in this decoding example are boxed.

The main problem with this partial decoding approach is that the number of required tables may be prohibitively large. A similar problem occurs in the Huffman case, which led to various attempts to reduce the number of tables [22, 24]. There is one table for each possible prefix of a codeword, so for Huffman codes, the number of tables is $N - 1$, where N is the size of the alphabet. This is true because a Huffman tree is a full binary tree, each prefix corresponds to an internal node, and the number of internal nodes is one less than the number of leaves. The Fib$m$ codes are also prefix codes, but the corresponding binary trees are not full, i.e., there are internal nodes having only one child.

An upper bound for the number of prefixes can be obtained as follows: let ℓ be the length of the longest of the N codewords, that is, using eq. (1), $N \leq \sum_{i=0}^{\ell-m} F_i^{(m)}$, the number of prefixes is then bounded by $\sum_{i=0}^{\ell-m}(i+m-1)F_i^{(m)}$. This bound is not tight, as prefixes shared by more than one codeword are counted more than once. Every codeword, except the first, of Fib$m$ is of the form $\alpha 01^m$, where α is some binary string, so that the set of different prefixes includes at least the strings $\alpha 01^j$, for $0 \leq j < m$. The number of tables is thus at least $mN - 1$. For example, for $m = 3$ and $N = 4$, the codewords are 111, 0111, 00111 and 10111, and the set of different prefixes is $\{1, 11, 0, 01, 011, 00, 001, 0011, 10, 101, 1011\}$. For $m = 3$ and an alphabet of $N = 100000$ codewords, even if each table entry needs only 4 bytes and one uses a low value of k such as $k = 8$, the space for the tables would be more than 300MB. The partial decoding approach is thus not feasible for the large values of N for which the Fib$m$ encoding schemes are intended as alternatives to the dense codes, but it could be an attractive variant for smaller alphabets.

3.3.2. Reducing the number of tables

It is possible to reduce the number of required tables using the properties of the Fibonacci numeration systems on which the Fib$m$ codes are based. Consider

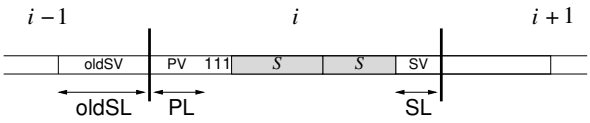

FIGURE 4: *Decoding a single byte*

the i-th byte of the text to be decoded, schematically represented in Figure 4. The byte can be partitioned into three zones: there is first a prefix of length PL, containing the bits necessary to complete a codeword, the first bits of which appeared already in the previous byte. This prefix may be empty (in case the previous codeword ended at the last bit of byte $i-1$), or it may, in the other extreme case, extend to the end of the i-th byte and possibly even beyond. Whether a codeword ends within the byte or not can be decided by the (non-)occurrence of m consecutive 1's. The second zone consists of zero or more complete codewords, depicted in grey in Figure 4. The third zone is the suffix of length SL of byte i and contains the prefix of a codeword, the completion of which appears only in the next byte or even later; again, this zone may be empty.

The partial decoding tables deal easily with the full codewords of the second zone and defer the treatment of the suffix in the third zone to the next iteration. The main problem is thus to relate the PL bits of the first zone to the oldSL bits of the third zone of the previous byte, so as to evaluate the index of this codeword that is split by the byte boundary. Contrarily to the decoding of Huffman codes, for which there is no connection between different parts of a codeword, the codewords of Fib$m$ codes are in fact representations of integers, which can be exploited as follows.

For the ease of description, we restrict the discussion and the examples in the sequel to the case $m = 3$, but all the statements can be easily generalized to other values of m. The codewords of Fib3 are defined as $C_1 = 111$, and for $i > 1$, $C_i = d_1 d_2 \cdots d_s 0111$, where $rel\_index = \sum_{j=1}^{s} d_j F_j^{(3)}$ is the relative index of the codeword C_i within the ordered set of codewords of length $s+4$; the absolute index is given as $i = rel\_index + \mathsf{Start}[s+4]$, where $\mathsf{Start}[\ell]$ is the index of the first codeword of length ℓ, as mentioned above in the description of the naive decoding algorithm.

Suppose the codeword $C_i = d_1 d_2 \cdots d_s 0111$ is split after the t-th bit into two parts that are treated separately: $D = d_1 d_2 \cdots d_t$ and $E = d_{t+1} \cdots d_s 0111$.

Denote by DV the value of the string D, that is, $DV = \sum_{j=1}^{t} d_j F_j^{(3)}$, and by EV the value of the string E from which the terminating 0111 has been stripped, considered independently of D, so that $EV = \sum_{j=t+1}^{s} d_j F_{j-t}^{(3)}$. For example, if $C_i = 011011011\mathbf{0111}$ and $t = 5$, we get $D = 01101$ and $E = 1011\mathbf{0111}$, and the corresponding values are $DV = 19$ and $EV = 12$. To recover the relative index, $V = 273$, of the original string within the sequence of codewords of length 13, EV has first to be "shifted" so that the bits of E correspond to the Fibonacci numbers $F_{t+1}^{(3)}, F_{t+2}^{(3)}, \ldots, F_s^{(3)}$, rather than to $F_1^{(3)}, F_2^{(3)}, \ldots, F_{s-t}^{(3)}$ as given in the definition of EV. But here we can use the fact that $F_j^{(3)} = [a_{(3)}\phi_{(3)}^j]$, so that roughly

$$F_{t+j}^{(3)} \simeq \phi_{(3)}^t \times F_j^{(3)}.$$

One would thus expect the requested value to be $V = DV + EV \times \phi_{(3)}^t$, rounded to the nearest integer. However, in our example, $DV + EV \times \phi_{(3)}^t = 271.60$, so even after rounding the target value would be missed by 1. This can be explained by the fact that though it is true that the j-th Fib3 number can be obtained by raising $\phi_{(3)}$ to the power j, multiplying by $a_{(3)}$ and then rounding, this property is not additive: when substituting, in our approximation, $F_j^{(3)}$ by $a_{(3)}\phi_{(3)}^j$, the *cumulative* error for several such substitutions is not necessarily smaller than 1/2, so that even after rounding, one may get wrong values.

To salvage the evaluation procedure, the approximate values should be used already in DV and EV. More precisely, instead of defining $EV = \sum_{j=t+1}^{s} d_j F_{j-t}^{(3)}$, let us define $EV = \sum_{j=t+1}^{s} d_j \phi_{(3)}^{j-t}$, and similarly for DV, without rounding to an integer value. The rounding will only be performed at the end, after having "shifted" EV by multiplying it with $\phi_{(3)}^t$. For our example, one gets thus $DV = 18.958$ and $EV = 12.063$, so that $DV + EV \times \phi_{(3)}^t = 272.881$; now the rounding gives the correct answer.

Summarizing, the revised version of the partial decoding tables still uses tables of 2^k entries for each of the possible k-bit bytes, each entry including the following information:

(i) PL: the length of the prefix of the byte terminating a codeword, not including the final 111, or if no codeword ends in this byte (no occurrence of 111), then PL= k;
(ii) PV: the approximate value of this prefix, using $\phi_{(3)}$;
(iii) S: a sequence of zero or more full codewords that could be decoded from the byte;
(iv) SL: the length of the suffix of the byte that has not be decoded;
(v) SV: the approximate value of this suffix, as above;
(vi) p: number of rightmost 1-bits in the SL-bit suffix of the byte, $0 \le p < m$.

```
1  (oldSL, oldSV, p)  ⟵  (0, 0, 0)
2  for i  ⟵  1  to length of text in bytes
3      (PL, PV, S, SL, SV, p)  ⟵  Tab[Text[i], p]
4      if PL = k      /* no codeword ends in this byte */
5          oldSV  ⟵  oldSV + PV × Phi[oldSL]
6          oldSL  ⟵  oldSL + PL
7      else
8          index  ⟵  [oldSV + PV × Phi[oldSL]]
                   +  Start[3 + oldSL + PL]
9          output C_index, S
10         (oldSV, oldSL)  ⟵  (SV, SL)
```

FIGURE 5: *Revised decoding procedure with partial decoding tables*

The last item, p, is needed to decide where the separating 111 strings occur. Indeed, suppose a byte starts with the bits 10111; if the previous byte ended with $p = 2$ non-decoded 1-bits, then only the leading 1 of the current byte is needed to complete a codeword and the byte contains, in addition, the codeword 0111; if the previous byte had $p < 2$, then the 5 first bits of the current byte are used to complete a codeword. Similarly, if in the previous byte $p = 1$, the current byte needs special treatment if it starts with 11. The decoding process thus uses three tables, one for each possible value of p.

The formal code is given in Figure 5. The procedure uses a precomputed array Phi, defined by Phi$[i] = \phi_{(3)}^i$. oldSL and oldSV hold the SL and SV values of the previous byte. The if-clause in lines 4–6 deals with the special case when no codeword ends in this byte (there is no occurrence of the substring 111), that is, the current codeword started either at the beginning of this byte or even earlier and extends into the next byte(s). In this case, one has only to update the length and value of the prefix of the current codeword. Otherwise, the index of the first codeword in this byte is evaluated in line 8.

A special case has to be dealt with when the terminating 111 of a codeword is split by a byte boundary, that is, for the previous byte $p \ge 1$, and the current byte starts with at least $3-p$ ones. PL has been defined as the length of the prefix in the first zone, not including the final 111, so to be consistent in our special case, PL should be defined as $-p$. This also corrects the index to be used in the Start table, since in the previous byte, all the oldSL bits were used to calculate the value oldSV, and only after reading the current byte did it turn out that the last p bits of the previous byte were a part of the separator 111, and thus should not participate in evaluating oldSV. Moreover, the value of PV should be zero in this case, but a correction term is needed to subtract the weight added by these last p bits to oldSV. If $p = 1$, the erroneously added amount is $a_{(3)}\phi_{(3)}^{\mathsf{oldSL}}$, so if one defines PV= $-a_{(3)}$, the definition

index		$p=0$						$p=1$						$p=2$					
dec	bin	PL	PV	S	SL	SV	p	PL	PV	S	SL	SV	p	PL	PV	S	SL	SV	p
0	00000000	8	0	–	0	0	0	8	0	–	0	0	0	8	0	–	0	0	0
1	00000001	8	80.9989	–	0	0	1	8	80.9989	–	0	0	1	8	80.9989	–	0	0	1
⋮	⋮																		
124	01111100	1	0	–	4	3.2296	0	1	0	–	4	3.2296	0	1	0	–	4	3.2296	0
125	01111101	1	0	–	4	10.3071	1	1	0	–	4	10.3071	1	1	0	–	4	10.3071	1
126	01111110	1	0	C_1	1	0	0	1	0	C_1	1	0	0	1	0	C_1	1	0	0
127	01111111	1	0	C_1	1	1.1374	1	1	0	C_1	1	1.1374	1	1	0	C_1	1	1.1374	1
128	10000000	8	1.1374	–	0	0	0	8	1.1374	–	0	0	0	-2	-1.474	–	7	0	0
129	10000001	8	82.1363	–	0	0	1	8	82.1363	–	0	0	1	-2	-1.474	–	7	44.0382	1
130	10000010	8	45.1757	–	0	0	0	8	45.1757	–	0	0	0	-2	-1.474	–	7	23.9431	1
131	10000011	8	126.175	–	0	0	2	8	126.175	–	0	0	2	-2	-1.474	–	7	67.9813	2
⋮	⋮																		
183	10110111	5	12.399	–	0	0	0	5	12.399	–	0	0	0	-2	-1.474	C_{15}	0	0	0
184	10111000	2	1.1374	–	3	0	0	2	1.1374	–	3	0	0	-2	-1.474	C_2	3	0	0
185	10111001	2	1.1374	–	3	3.8480	1	2	1.1374	–	3	3.8480	1	-2	-1.474	C_2	3	3.8480	1
⋮	⋮																		
220	11011100	3	3.2296	–	2	0	0	-1	-.6184	C_2	2	0	0	-2	-1.474	C_4	2	0	0
221	11011101	3	3.2296	–	2	2.0921	1	-1	-.6184	C_2	2	2.0921	1	-2	-1.474	C_4	2	2.0921	1
222	11011110	3	3.2296	–	2	1.1374	0	-1	-.6184	C_2	2	1.1374	0	-2	-1.474	C_4	2	1.1374	0
223	11011111	3	3.2296	–	2	3.2296	2	-1	-.6184	C_2	2	3.2296	2	-2	-1.474	C_4	2	3.2296	2
224	11100000	0	0	–	5	0	0	-1	-.6184	–	6	1.1374	0	-2	-1.474	–	7	3.2296	0
225	11100001	0	0	–	5	13.0176	1	-1	-.6184	–	6	25.0805	1	-2	-1.474	–	7	47.2678	1
226	11100010	0	0	–	5	7.0775	0	-1	-.6184	–	6	14.1550	0	-2	-1.474	–	7	27.1726	0
227	11100011	0	0	–	5	20.0951	2	-1	-.6184	–	6	38.0981	2	-2	-1.474	–	7	71.2108	2
⋮	⋮																		
254	11111110	0	0	C_1	2	1.1374	0	-1	-.6184	C_1	3	3.2296	0	-2	-1.474	C_1C_1	1	0	0
255	11111111	0	0	C_1	2	3.2296	2	-1	-.6184	C_1C_1	0	0	0	-2	-1.474	C_1C_1	1	1.1374	1

TABLE 5: *Sample lines of revised partial decoding tables*

precision	$a_{(3)}$	$\phi_{(3)}$	index of first wrong codeword
32	0.6184199	1.8392868	189,473
64 / 32			2,097,155
64	0.6184199223193926	1.8392867552141611	—

TABLE 6: *Influence of precision on the correctness*

of *index* in line 8 applies also in this case. For $p = 2$, the amount to be subtracted is

$$\begin{aligned} a_{(3)}\phi_{(3)}^{\mathsf{oldSL}-1} + a_{(3)}\phi_{(3)}^{\mathsf{oldSL}} &= a_{(3)}\left(1+\tfrac{1}{\phi_{(3)}}\right)\phi_{(3)}^{\mathsf{oldSL}} \\ &= 1.4738\ \phi_{(3)}^{\mathsf{oldSL}}, \end{aligned}$$

so PV should be defined as -1.4738. Table 5 brings some sample lines of the revised partial decoding tables for $k = 8$ and $m = 3$.

To estimate the size of the tables, consider first $k = 8$. Just 4 bits are then needed for the PL and SL fields, and 2 bits for p (which is enough also for $m = 4$). S will store the index of the decoded codeword rather than the codeword itself, so the number of bits allocated to S depends on the size N of the alphabet. If $N < 4$ million, and in most cases it will be, one can pack S and p together in 3 bytes. Of course, there is the possibility of more than one codeword being decoded from a single byte (see, e.g., the last line in Table 5), but for $k = 8$, this can only happen for the pair C_1C_1. Instead of reserving space for two codewords within the string S, one rather may deal with this special case by treating the pair C_1C_1 as if it were another symbol, indexed, e.g., $N+1$. The values PV and SV are stored as 4-byte float numbers. Adding it up, one needs 12 bytes per entry, so for $m = 3$ and $k = 8$, the total size is $2^8 \times 3 \times 12$, less than 10K. Such a low value suggests that it might even be reasonable to use $k = 16$, processing twice as many bits in each iteration. The size of each entry may increase to 13 bytes (there are more possible values for PL and SL, and more special cases have to be taken care of if one restricts S to a single index), but the overall

size is still $2^{16} \times 3 \times 13$, less than 2.5MB, which is often acceptable.

It should be noted that the validity of the above decoding procedure depends on the precision used to represent floating point numbers in the program. If single precision is used (32 bits of which 23 bits are the mantissa), the accumulated error when raising $\phi_{(3)}$ to the power j will yield wrong values for $j \geq 24$. For a given number i, a growing number of different Fibonacci numbers is used in its representation, and here again the absolute value of the error introduced by lower precision adds up until it may exceed 0.5, so that rounding will not give the required integer. We have checked all the numbers up to the first error and found that it occurs for $i = 189473$. If the alphabet to be dealt with is smaller, then single precision is enough. However, it seems that for larger alphabets (such as all the examples in Table 3), double precision (64 bits, 52 bit mantissa) must be used. But double precision will not only increase substantially the size of the tables, it is also much more time consuming. A compromise could be a hybrid approach, using double precision only for all the off-line evaluations, that is, for the all values in the tables. Thus up to $\phi_{(3)}^8$, we shall use double precision to calculate the value, but actually store them in the table in four bytes only; all the online evaluations (lines 5 and 8) are done with single precision. Table 6 brings the indices of the first wrong elements for the given precision, as well as the according values of $a_{(3)}$ and $\phi_{(3)}$. The middle line, headed 64/32, corresponds to the hybrid case. For double precision, we checked all the values up to $F_{39}^{(3)} > 12.96$ billion, and did not find any rounding error. In all these cases, $-0.392 < error < 0.247$, where *error* is the difference between the correct (integer) value and the value based on powers of $\phi_{(3)}$.

3.3.3. Eliminating multiplications

One may still object that the use multiplications, floating point numbers and rounding can have a negative impact on the processing time. One can get rid of all these at the cost of some additional tables. The idea is to replace the multiplications with Phi[oldSL] in lines 5 and 8 by pre-calculated values of shifted Fibonacci numbers. One thus prepares a two-dimensional table Fib3[*index*, *shift*], *index* running over all the possible PV values, and *shift* over the possible shifts, that is, from zero to the length of the longest possible prefix of a codeword. According to our earlier notation, the bitstring $x_1 \cdots x_k$ represents the integer $\sum_{i=1}^{k} x_i F_i^{(3)}$. Fib3[*index*, *shift*] will contain the corresponding value obtained by shifting all the bits by *shift* bits to the right, i.e.,

$$\mathsf{Fib3}[index, shift] = \sum_{i=1}^{k} x_i F_{i+shift}^{(3)}.$$

index dec	Fib-bin	*shift* 0	1	2	···	10	···
⋮	⋮						
56	10110010	56	103	190		24858	
57	00001010	57	105	193		25281	
58	10001010	58	107	197		25785	
59	01001010	59	109	200		26208	
60	11001010	60	111	204		26712	
61	00101010	61	112	206		26986	
⋮	⋮						
147	01011011	147	271	498		65234	
148	11011011	148	273	502		65738	
149	$p = 1$	–	-1	-2		-274	
150	$p = 2$	–	–	-3		-423	

TABLE 7: *Sample lines and columns of the* Fib3[*index*, *shift*] *table*

Table 7 displays some sample lines and columns of the Fib3[*index*, *shift*] table. The last two lines are a special case, to be explained below.

Using this table, the algorithm of Figure 5 can be modified by replacing lines 5 and 8, respectively, by

5 oldSV ⟵ oldSV + Fib3[PV, oldSL]

and

8 *index* ⟵ oldSV + Fib3[PV, oldSL]
+ Start[3 + oldSL + PL]

Note that all the values are integers now, so no rounding is necessary and there are no multiplications. The tables Tab[] can be simplified and all float numbers be replaced by the corresponding integers, using for each only 1 instead of 4 bytes. To adapt the modified algorithm also to the special cases when the separating 111 is split by byte boundaries, note that when $p = 1$ and the current codeword starts with 1, then the rightmost 1-bit of the previous byte contributed the amount of $F_{\mathsf{oldSL}}^{(3)}$ to oldSV, but this addition was erroneous and should now be corrected. For $p = 2$, in case the current codeword starts with 1, the erroneous addition is $F_{\mathsf{oldSL}-1}^{(3)} + F_{\mathsf{oldSL}}^{(3)}$.

As above, the correct value of PL in these cases is $-p$. The correction factors will be accommodated in the Fib3 table, since it is used in lines 5 and 8. Two new lines are added to the tables, indexed $M+1$ and $M+2$, where M is the index of the highest of the PV values. For $k = 8$, the highest number that can be represented in the Fib3 representation is 11011011, corresponding to $M = 148$ and for $k = 16$, the largest number is 1011011011011011, representing $M = 19511$. The new entries are defined as

$$\begin{aligned} \mathsf{Fib3}[M+1, s] &= -F_s^{(3)} \\ \mathsf{Fib3}[M+2, s] &= -F_{s-1}^{(3)} - F_s^{(3)}. \end{aligned}$$

All that remains to be done is then to define, in the Tab tables, the value PV as $M + p$ in case PL$= -p$. Table 8 shows a few sample lines of the updated Tab tables.

index		$p=0$						$p=1$						$p=2$					
dec	bin	PL	PV	S	SL	SV	p	PL	PV	S	SL	SV	p	PL	PV	S	SL	SV	p
⋮	⋮																		
124	01111100	1	0	–	4	3	0	1	0	–	4	3	0	1	0	–	4	3	0
125	01111101	1	0	–	4	10	1	1	0	–	4	10	1	1	0	–	4	10	1
126	01111110	1	0	C_1	1	0	0	1	0	C_1	1	0	0	1	0	C_1	1	0	0
127	01111111	1	0	C_1	1	1	1	1	0	C_1	1	1	1	1	0	C_1	1	1	1
128	10000000	8	1	–	0	0	0	8	1	–	0	0	0	-2	149	–	7	0	0
129	10000001	8	82	–	0	0	1	8	82	–	0	0	1	-2	149	–	7	44	1
⋮	⋮																		
222	11011110	3	3	–	2	1	0	-1	150	C_2	2	1	0	-2	149	C_4	2	1	0
223	11011111	3	3	–	2	3	2	-1	150	C_2	2	3	2	-2	149	C_4	2	3	2
224	11100000	0	0	–	5	0	0	-1	150	–	6	1	0	-2	149	–	7	3	0
225	11100001	0	0	–	5	13	1	-1	150	–	6	25	1	-2	149	–	7	47	1
⋮	⋮																		

TABLE 8: *Sample lines of updated, integer only, partial decoding tables*

The reduction in processing time and in the size of the Tab tables comes at the price of storing the additional Fib3 tables, the number of which depends on the size of the alphabet, N. There is one table for each possible shift, and the number of shifts is the size in bits of the largest integer, which is 16, 24 and 28 for $N = 10^4$, 10^5 and 10^6, respectively.

Table 9 summarizes the space needed for the various decoding methods for different sizes N of the alphabet, and for $k = 8$ or 16. Full tables refers to the basic partial decoding tables of Section 3.3.1 and the values for it are lower bounds. As the size is at least proportional to N, this method is not feasible for larger alphabets. The columns headed Mult correspond to the algorithm of Figure 5 using multiplications and floating point numbers in the tables. Their space does not depend on N. Finally, the last column for each value of N is for the method using in addition to the partial decoding tables, also the Fib3 tables. The additional space is $O(\log N)$.

As can be seen, the last two methods have reasonable space requirements, and for low values of N, the last method, which is also faster than the previous one, may even require less space. For larger alphabets, the revised partial decoding tables with floats or integers offer a time/space tradeoff.

To empirically compare timing results, we chose the following input files of different sizes and languages: the Bible (King James version) in English, and the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [25]. The dictionary of the Bible was generated after stripping the text of all punctuation signs, whereas the French text has not been altered and every string of consecutive non-blanks was considered as a dictionary term; for example, "`tour`", "`Tour`" and "`tour.`" are considered as different terms in French, but generate only one element in the English dictionary.

The results are presented in Table 10, which includes also relevant statistics, such as the size of the file in words and the number of different words. The timing results correspond to the last version of the decoding mentioned, that using the Fib3[*index*, *shift*] tables and the partial decoding tables as those in Table 8. Note that because of the preprocessing of the files, their sizes are different from standard versions of the Bible or the JOC corpus. The values in the column headed Compressed size are given in average number of bits per word, and the numbers in parentheses in the SCDC column give the optimal (s, c) pair used for the given distribution. The time values are in seconds and were averaged over 100 independent runs of the full decoding of each text on an IBM X366 e-series with 6GB RAM running Red Hat Enterprise Linux. The column headed Bit-Fib3 refers to the naive bitwise decoding, whereas the column headed Block-Fib3 is the table-driven variant described in this section, with $k = 8$. The programs for the SCDC decoding, as well as those used for the compressed search below, have kindly been provided by the authors of [9]. We see that the Fib3 files are 9-10% shorter than the optimal (s, c)-codes. Relative to the naive bitwise Fib3 decoding, the block based Fib3 approach is, on the given data, about 30% faster, but it still takes roughly twice as long as the decoding of the byte aligned SCDC codes.

3.4. Compressed search

The main motivation for the creation of Tagged Huffman codes was to support the possibility of searching directly within the compressed text. Regular Huffman codes, even byte-aligned ones, suffer from a synchronization problem. Consider, for example, the code $\{0, 11, 100, 101\}$ for the alphabet $\{A, B, C, D\}$, then the encoding of DC would be 101100, and the substring 11 in the middle could be falsely identified as

	$N = 10,000$			$N = 100,000$			$N = 1,000,000$		
	Full tables	Mult	with Fib3	Full tables	Mult	with Fib3	Full tables	Mult	with Fib3
$k = 8$	30M	10K	10K	300M	10K	15K	3G	10K	21.4K
$k = 16$	7.5G	2.5M	1.9M	75G	2.5M	2.6M	750G	2.5M	3.4M

TABLE 9: *Required storage space for different decoding methods*

Database	Total # of words	# different words	Compressed size Fib3	SCDC	Time (sec) Bit-Fib3	Block-Fib3	SCDC
Bible KJV	611,793	11,378	9.34	(223,33) 10.28	0.111	0.079	0.034
French	1,176,192	75,192	11.12	(188,68) 12.36	0.261	0.178	0.092

TABLE 10: *Empirical comparison of Fib3 and SCDC decoding*

the encoded form of B. If the search is done sequentially from the beginning of the text, one could record the codeword boundaries and then even a regular Huffman code could be used. But this would disqualify non-sequential searches, which can be much more efficient, such as the Boyer-Moore (BM) algorithm [26]. So if BM searches are to be supported, the synchronization problem has to be dealt with.

In the tagged Huffman version, the tag identifies the last byte of each codeword so that a tagged byte can *only* appear as the last byte of a codeword. Similarly, the last byte of a codeword for ETDC must have a value less than 128 and for SCDC less than s. In the bit-oriented Fib$m$ codes, the end of a codeword is detected by the appearance of a string of m consecutive 1s, which occur at the end of each codeword, and only there. However, for all four mentioned codes, if a given codeword $\mathcal{E}(x)$ is located in the compressed text $\mathcal{E}(T)$, it does not necessarily correspond to a real occurrence of the encoded item x in the text T, because all these codes have codewords that are suffixes of others. One therefore needs to check also the bits just preceding the located match: it is a true occurrence of x if the preceding byte is tagged for the tagged Huffman codes, or if the value of the preceding byte is $< s$ for SCDC.

For Fib$m$, the m bits preceding the match have to be all 1s, but even then one cannot be sure that a true match has been detected. For example, suppose one is looking for the word `day`, encoded by 1011000111 in the Fib3 compressed KJV Bible. If one uses this pattern, extended at its left end by 111, it will be erroneously located as a suffix of 100001000111-11011000111, which is the encoding of `burnt offering`. What causes the problem is the existence of codewords in Fib3 that start with one or two 1s.

One way to overcome this difficulty is to redefine the set of codewords of Fib3, eliminating the first codeword, 111, which is the only one not containing a zero. As a result, all the codewords have then 0111 as suffix, so to find $\mathcal{E}(x)$, one can look for $0111\mathcal{E}(x)$ in $0111\mathcal{E}(T)$. This would come with a penalty of increasing the lengths of certain codewords, yielding, respectively, an average length of 9.52 and 11.27 bits for the Bible and the French texts used above, an increase of less than 2%. This could be tolerable as the files would still be shorter by 7–9% than the corresponding optimal SCDC variants.

Database	SCDC	Fib3 (Byte)	Fib3 (Bit)
Bible KJV	0.88	2.48	81.7
French	0.67	2.04	65.5

TABLE 11: *Empirical comparison of compressed search (in milliseconds)*

As second option, when better compression is critical, the search procedure in the Fib3 encoded text can be amended similarly to the additional check of the byte preceding the match in tagged Huffman or SCDC. To locate $\mathcal{E}(x)$, one searches for the pattern $111\mathcal{E}(x)$, and if it is found starting at position i in the encoded text, one goes through the following sequence of checks: the match is declared a true occurrence of x if either the bit in position $i-1$ is 0, or the string terminating in position $i-1$ is 0111, or 0111111, or generally $0(111)^t$ for some $t \geq 0$. This seems as a potentially infinite sequence of checks, but practically, the appearance, for $t > 0$, of the string $0(111)^t$ ending at position $i-1$, means that the term x is preceded in the text by t copies of the most frequent term (encoded by 111), which is often the word `the`. In natural language texts it will be very rare to have two or more such consecutive occurrences. One can therefore restrict the check to, say, $t \leq 2$, at the risk of very rarely announcing a false match; or one could in a preparation phase evaluate the length of the longest sequence of consecutive occurrences of the most frequent term, and set t accordingly.

As a final option, the problem caused by the shortest codeword 111 can be circumvented by encoding all the (non-overlapping) occurrences of *pairs* of the most frequent term by a single codeword, rather than by two

i	Code	Decoded text
80	Fib3	In the beginning God created the heaven and the **before** And the earth was without form and void and darkness was ...
	Huf	In the beginning God created the heaven and the **father thy of upon** was without form and void and darkness was ...
	SCDC	In the beginning God created the **sin And of the Jacob against called thine mouth whom to hands with to sword** ...
160	Fib3	And the earth was without form and — darkness was upon the face of the deep. And the Spirit of God moved upon ...
	Huf	And the earth was without form and void and **therefore in the Tamar that and the was we shall** Spirit of God moved ...
	SCDC	And the earth was without **mouth whom to hands with to sword for but at and eat And and voice took his and** ...

TABLE 12: *Illustration of robustness: decoded text after having deleted the bit at position number i*

consecutive 111s. The phrase `the the` would thus be assigned a special codeword in many natural language texts, but the loss of compression efficiency would be negligible if such a phrase appears only very rarely, if at all. On the other hand, in the search procedure above, one can restrict the number of checks to $t = 1$.

Basically, a BM search in SCDC or tagged Huffman compressed text is faster, since it is performed byte per byte, whereas the Fib$m$ compressed texts have to be scanned on the bit level. However, in our tests, we used a byte-oriented version of the BM algorithm (Byte) described in [27], which reads and compares whole blocks, typically bytes, instead of manipulating individual bits. Table 11 reports the average time of sample runs, performed on the same test files as above. The patterns have been chosen as the encodings of single codewords of various lengths. Emulating the tests described in [9], we randomly chose terms of the database vocabulary of various lengths (from 5 to 21 characters), and considered the corresponding SCDC and Fib3 encodings. The SCDC encodings were 1 to 3 bytes long, and the lengths of the Fib3 encoded patterns were between 14 and 24 bits for French, and between 9 and 21 bits for English. For each of the sets, 700 terms were chosen and each run was repeated 100 times. The time measured was until the first occurrence of the binary SCDC or Fib3 encoding has been located in the SCDC or Fib3 compressed text, respectively, averaged for each language and over the 100 runs. For comparison, we also ran the regular bit-oriented BM algorithm (Bit). The results show that while the SCDC algorithm is almost 100 times faster than bitwise BM searching, it is just up to three times faster than the performance of byte oriented BM on our tests, giving a reasonable tradeoff between the compression ratio and the processing speed.

3.5. Robustness

As a final criterion, we turn to one that has not been mentioned in [4, 8], which is the robustness of the code against errors. If the codes are to be used over a noisy communication channel, their resilience to bit insertions, deletions and to bit-flips may be of high importance.

Similarly to the case of fixed length codes, the damage caused by a bit change in the SCDC codes will be locally restricted. One codeword will be changed into another of equal length or will be split into two codewords, and two adjacent codewords may fusion into a single one, but there will be no propagation beyond that. The insertion or deletion of even a single bit, on the other hand, may be devastating, rendering the suffix of the text following the error useless, as the decoding will be shifted and all the true codeword boundaries could be missed. The same may be true for variable length Huffman codes, though they generally have a tendency to resynchronize quickly [6]. In contrast, Fibonacci codes are robust even against insertions and deletions: since the codeword endings are explicitly given by the string of m 1s, and do not depend on the position, an error will affect only a single codeword, or at most two adjacent ones.

Table 12 brings a small example to illustrate the differences in robustness of some of the codes discussed above. The text is that of the KJV Bible, encoded as a sequence of words, and three codes were tested: Fib3, Huffword and the optimal SCDC with $(s, c) = (223, 33)$. To simulate the occurrence of an error—in our case, a bit getting lost—a single bit has been deleted (at position $i = 80$ or 160) for each of the files, and the resulting files were decoded using the corresponding regular decoding routines. Erroneously decoded words are boldfaced for emphasis.

As expected, only one or two codewords were lost for Fib3. For $i = 80$, the codeword 01010000111 (`earth`) turned into 0101000111 (`before`); for $i = 160$, the deleted bit was in the separating 111 string, so the two codewords 110100110110111 (`void`) and 0111 (`and`) fusioned to 110100110110110111, which represents the index number 12627; there are, however, only 11378 elements in the alphabet, so we have here an example of a case in which the error can be detected, even if there is no way to correct it. Note that for Huffword and SCDC, even the fact that an error has occurred can go undetected.

Huffman coding provides interesting examples: one could have expected that the error might propagate, but in fact, after 4–10 codewords, synchronization is regained in these examples, and this is also generally the case, as mentioned in [6]. Finally, for SCDC, all the

codewords after the error are shifted one bit to the left, so the decoding will never resynchronize, and the error will be detected only at the end of the encoded string.

4. CONCLUSION

The paper suggests the use of higher order Fibonacci codes as efficient alternatives to recently proposed compression codes such as tagged Huffman codes, End-Tagged dense codes and (s, c)-dense codes. It shows that on some criteria, including compression efficiency and the robustness against errors, Fibonacci codes may be preferable and perform better than the other codes, while still being inferior on decompression and compressed search speed. Overall, the Fibonacci codes may thus give a plausible tradeoff for certain applications.

For the cases in which Fibonacci codes have been chosen, independently of their competition with the other alternatives, we have shown how to accelerate their decoding by means of decoding tables that have been prepared in advance and depend only on the code, not on the given encoded text.

REFERENCES

[1] Manber, U. (1997) A text compression scheme that allows fast searching directly in the compressed file, ACM Trans. on Inf. Sys., 15, 124–136.

[2] Farach, M. and Thorup, M. (1998) String matching in Lempel-Ziv compressed strings, Algorithmica, 20(4), 388–404.

[3] de Moura, E.S., Navarro, G., Ziviani, N. and Baeza-Yates R. (2000) Fast and flexible word searching on compressed text, ACM Trans. on Information Systems, 18, 113–139.

[4] Brisaboa, N.R., Fariña, A., Navarro, G. and Paramá J.R. (2005) Efficiently decodable and searchable natural language adaptive compression, Proc. 28-th ACM-SIGIR Conference, Salvador, Brazil, August 15–19, 234–241. ACM, New York.

[5] Moffat, A. (1989) Word-based text compression, Software – Practice & Experience, 19, 185–198.

[6] Klein, S.T. and Shapira, D. (2005) Pattern matching in Huffman encoded texts, Information Processing & Management, 41(4), 829–841.

[7] Biskup, M.T. (2008) Guaranteed synchronization of Huffman codes, Proc. Data Compression Conference DCC–2008, Snowbird, Utah, March 15–17, 462–471. IEEE Computer Society, Los Alamitos, CA.

[8] Brisaboa, N.R., Iglesias, E.L., Navarro, G. and Paramá J.R. (2003) An efficient compression code for text databases, Proc. European Conference on Information Retrieval ECIR'03, Pisa, Italy, April 14–16, LNCS 2633, 468–481. Springer Verlag, Berlin.

[9] Brisaboa, N.R., Fariña, A., Navarro, G. and Esteller M.F. (2003) (S,C)-dense coding: an optimized compression code for natural language text databases, Proc. Symposium on String Processing and Information Retrieval SPIRE'03, Manaus, Brazil, October 8–10, LNCS 2857 122–136. Springer Verlag, Berlin.

[10] Apostolico, A. and Fraenkel, A.S. (1987) Robust transmission of unbounded strings using Fibonacci representations, IEEE Trans. Inform. Theory, 33, 238–245.

[11] Fraenkel, A.S. and Klein, S.T. (1996) Robust universal complete codes for transmission and compression, Discrete Applied Mathematics, 64, 31–55.

[12] Przywarski, R., Grabowski, S., Navarro, G. and Salinger, A. (2006) FM-KZ: an even simpler alphabet-independent FM-index, Proc. Prague Stringology Conference, Prague, Czech Republic, August 28-30, 226–240. Czech Technical University, Prague.

[13] Lelewer, D.A. and Hirschberg, D.S. (1987) Data compression, ACM Computing Surveys, 19, 261–296.

[14] Klein, S.T. (2000) Skeleton trees for the efficient decoding of Huffman encoded texts, Kluwer Journal of Information Retrieval, 3, 7–23.

[15] Liddell, M. and Moffat, A. (2006) Decoding prefix codes, Software—Practice and Experience, 36, 1687–1710.

[16] Bookstein, A. and Klein, S.T. (1993) Is Huffman coding dead?, Computing, 50, 279–296.

[17] Moffat, A., Zobel, J. and Sharman, N. (1997) Text compression for dynamic document databases, IEEE Transactions on Knowledge and Data Engineering, 9, 302–313.

[18] Zipf, G.K. (1935) *The Psycho-Biology of Language*, Houghton Mifflin, Boston, MA.

[19] Bookstein, A., Klein, S.T. and Ziff, D.A. (1992) A systematic approach to compressing a full text retrieval system, Information Processing & Management, 28, 795–806.

[20] Fraenkel, A.S. (1976) All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, Jurimetrics J., 16, 149–156.

[21] Brisaboa, N.R., Fariña, A., Navarro, G. and Paramá J.R. (2007) Lightweight natural language text compression, Information Retrieval, 10(1), 1–33.

[22] Choueka, Y., Klein, S.T. and Perl Y. (1985) Efficient variants of Huffman codes in high level languages, Proc. 8-th ACM-SIGIR Conf. Montreal, Canada, June 5–7, 122–130. ACM, New York.

[23] Sieminski, A. (1988) Fast decoding of Huffman codes, Information Processing Letters, 26, 237–241.

[24] Bergman, E. and Klein, S.T. (2005) Fast decoding of prefix encoded texts, Proc. Data Compression Conference DCC–2005, Snowbird, Utah, March 29–31, 143–152. IEEE Computer Society, Los Alamitos, CA.

[25] Véronis, J. and Langlais, P. (2000) Evaluation of parallel text alignment systems: The arcade project, Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 369–388.

[26] Boyer, R.S. and Moore, J.S. (1977) A fast string searching algorithm, Communications of the ACM, 20, 762–772.

[27] Klein, S.T. and Kopel Ben-Nissan M. (2007) Accelerating Boyer Moore searches on binary texts, Proc. Intern. Conf. on Implementation and Application of Automata, CIAA-07, Prague, Czech Republic, July 16-18, LNCS 4783, 130–143. Springer Verlag, Berlin.