On the Usefulness of Fibonacci Compression Codes

SHMUEL T. KLEIN* AND MIRI KOPEL BEN-NISSAN

Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
*Corresponding author: tomi@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.

Keywords: text compression; Fibonacci codes; dense codes

Received 7 June 2008; revised 29 December 2008
Handling editor: Alberto Apostolico

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 [1–4] 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]. However, 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 an 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 that 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 satisfies \( b_r \leq s \) 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 \( F^{(m)}_i \), are defined by the following recurrence relation:

\[
F^{(m)}_n = F^{(m)}_{n-1} + F^{(m)}_{n-2} + \cdots + F^{(m)}_{n-m} \quad \text{for } n > 0,
\]

and the boundary conditions

\[
F^{(m)}_0 = 1 \quad \text{and} \quad F^{(m)}_n = 0 \quad \text{for } -m < n < 0.
\]

For fixed order \( m \), the number \( F^{(m)}_n \) can be represented as a linear combination of the \( n \)th 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 \( > 1 \), 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 \( F^{(m)}_n \) 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 \( \phi(2) = (1 + \sqrt{5})/2 = 1.6180 \) is the well-known golden ratio. As a matter of fact, the entire Fibonacci sequence can be obtained by \( F^{(m)}_n = [a(m)\phi^{(m)}_{n}] \), where \( a(m) \) is the coefficient of the dominating term in the above-mentioned linear combination, and \([x]\) indicates that the value of the real number \( x \) is rounded to the closest integer, that is, \([x] = [x + 0.5]\). 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 \( \phi^{(m)}_{n} \). For larger \( n \), the numbers \( a(m)\phi^{(m)}_{n} \) are usually quite close to integers, e.g. \( a(2)\phi^{(10)}_{(2)} = 88.998 \) and \( a(3)\phi^{(13)}_{(3)} = 1705.01 \).

The standard representation of an integer as a binary string is based on a notation 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 \( B = \sum_{i=0}^{k-1} b_i 2^i \). However many other possible binary representations do exist, and those using the Fibonacci sequences as basis elements have some interesting properties. We 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 \( B = \sum_{i=1}^{r} c_i F^{(2)}_i \). The representation will be unique if one uses the following procedure to produce it: given the integer \( B \), find the largest Fibonacci number \( F^{(2)}_i \) smaller or equal to \( B \); then continue recursively with \( B - F^{(2)}_i \). For example, 45 is 34 + 8 + 3, so that 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 \([u_1, u_2, \ldots] = \{11, 110, 1100, 11001, 110001, \ldots\} \), where the separating 1 is put in boldface for visibility. A typical compressed text could be 1100111001101111101, which is easily parsed as \( u_4 u_3 u_2 u_1 u_2 \). Though being UD, this is not a prefix code, and 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 110111111110; a first attempt to parse it as 110 11 11 11 10 = \( u_2 u_1 u_1 u_1 \) 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 11 11 = \( u_4 u_3 u_2 u_1 u_2 \).

To overcome this problem, [10, 11] suggest to reverse all the codewords, yielding the set \([v_1, v_2, \ldots] = \{11, 011, 0011, 1011, 00011, 10011, \ldots\} \), which is a prefix 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. 10111 precedes 01011.

The generalization to higher order seems at first sight straightforward: any integer \( B \) can be uniquely represented by the string \( d_md_{m-1} \cdots d_1 \) such that \( B = \sum_{i=1}^{m} d_i F_i^{(m)} \) using the iterative encoding procedure mentioned above. In this representation, there are no consecutive substrings of 1s. For example, the representations of the integers 10, 11, 12 and 13 using \( F^{(3)} \) are, respectively, 1011, 1100, 1101 and 10000. However simply adding now \( m-1 \) 1s 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 \( \{v_{10}, \ldots, v_{13}\} = \{110111, 001111, 101111, 0000111\} \), but the encoding of the fourth element of the sequence would be \( v_4 = 001111 \), which is a prefix of \( v_11 \). The string 00111110111 could be parsed both as 00111110111 and as 00111111111.

The problem stems from the fact that for \( m > 2 \), there can be more than one leading 1 in the representation of an integer, and hence adding \( m-1 \) 1s may give a string of up to \( 2m-2 \) consecutive 1s. The fact that a string of 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\( n \) 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 \( F_i^{(2)} \); 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 \( n \geq 0 \), the code Fib\( m \) consists of

\[
F_n^{(m)} \text{ codewords of length } n + m.
\]

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

---

### Table 1. Fibonacci numbers of order \( m = 2, 3, 4, 5, 6 \).

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

---

### Table 2. Fibonacci codes of order \( m = 2, 3, 4 \).

<table>
<thead>
<tr>
<th>Index</th>
<th>Fib2</th>
<th>Fib3</th>
<th>Fib4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>111</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
<td>0111</td>
<td>01111</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>00111</td>
<td>001111</td>
</tr>
<tr>
<td>4</td>
<td>1011</td>
<td>10111</td>
<td>101111</td>
</tr>
<tr>
<td>5</td>
<td>00111</td>
<td>000111</td>
<td>0001111</td>
</tr>
<tr>
<td>6</td>
<td>10011</td>
<td>100111</td>
<td>1001111</td>
</tr>
<tr>
<td>7</td>
<td>01011</td>
<td>010111</td>
<td>0101111</td>
</tr>
<tr>
<td>8</td>
<td>000111</td>
<td>0000111</td>
<td>00001111</td>
</tr>
<tr>
<td>9</td>
<td>100111</td>
<td>1000111</td>
<td>10001111</td>
</tr>
<tr>
<td>10</td>
<td>010111</td>
<td>0100111</td>
<td>01001111</td>
</tr>
<tr>
<td>11</td>
<td>001111</td>
<td>0001111</td>
<td>00011111</td>
</tr>
<tr>
<td>12</td>
<td>101111</td>
<td>1010111</td>
<td>10101111</td>
</tr>
<tr>
<td>13</td>
<td>0001111</td>
<td>00001111</td>
<td>000011111</td>
</tr>
<tr>
<td>14</td>
<td>1000111</td>
<td>10001111</td>
<td>100011111</td>
</tr>
<tr>
<td>15</td>
<td>0100111</td>
<td>01001111</td>
<td>010011111</td>
</tr>
<tr>
<td>16</td>
<td>0010111</td>
<td>00010111</td>
<td>000101111</td>
</tr>
<tr>
<td>17</td>
<td>1010111</td>
<td>10010111</td>
<td>100101111</td>
</tr>
<tr>
<td>18</td>
<td>00010111</td>
<td>000010111</td>
<td>0000101111</td>
</tr>
<tr>
<td>19</td>
<td>10010111</td>
<td>100001111</td>
<td>1000011111</td>
</tr>
<tr>
<td>20</td>
<td>01010111</td>
<td>001010111</td>
<td>0010101111</td>
</tr>
<tr>
<td>21</td>
<td>000010111</td>
<td>0000010111</td>
<td>00000101111</td>
</tr>
<tr>
<td>22</td>
<td>100010111</td>
<td>1000010111</td>
<td>10000101111</td>
</tr>
<tr>
<td>23</td>
<td>010010111</td>
<td>0100010111</td>
<td>01000101111</td>
</tr>
<tr>
<td>24</td>
<td>001010111</td>
<td>0001010111</td>
<td>00010101111</td>
</tr>
<tr>
<td>25</td>
<td>101010111</td>
<td>1010010111</td>
<td>10100101111</td>
</tr>
<tr>
<td>26</td>
<td>001010111</td>
<td>0001010111</td>
<td>00010101111</td>
</tr>
<tr>
<td>27</td>
<td>101010111</td>
<td>1010010111</td>
<td>10100101111</td>
</tr>
<tr>
<td>28</td>
<td>0101010111</td>
<td>00101010111</td>
<td>001010101111</td>
</tr>
<tr>
<td>29</td>
<td>0001010111</td>
<td>00001010111</td>
<td>000010101111</td>
</tr>
<tr>
<td>30</td>
<td>1001010111</td>
<td>100001010111</td>
<td>1000010101111</td>
</tr>
<tr>
<td>31</td>
<td>0101010111</td>
<td>010001010111</td>
<td>0100010101111</td>
</tr>
<tr>
<td>32</td>
<td>00101010111</td>
<td>000101010111</td>
<td>0001010101111</td>
</tr>
<tr>
<td>33</td>
<td>10101010111</td>
<td>101001010111</td>
<td>1010010101111</td>
</tr>
<tr>
<td>34</td>
<td>00000010111</td>
<td>0000001010111</td>
<td>00000010101111</td>
</tr>
<tr>
<td>35</td>
<td>100000010111</td>
<td>10000001010111</td>
<td>100000010101111</td>
</tr>
</tbody>
</table>
We now turn to an investigation of some properties of the Fibonacci 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 [8,9] 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 simple as equivalent on the second and inferior on the last two. A simple implementation of decoding and searching could be as simple as above 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 a 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. Therefore 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. However the assumption here, like in [8,9], is that one chooses a single code that 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 bitwise 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]. However 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 bit string.

3.1. Compression efficiency

We 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 \(\phi^k\). 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 \(128^{M} - 128^{M-1} = 2^M\). For Fib\(m\), the last \(m\) bits of each codeword are reserved; hence using equation (1), the number of codewords with up to 8\(M\) bits is approximately

\[
\sum_{i=1}^{8M-8} F_i(a(m)) \approx \sum_{i=1}^{8M-8} \phi_i(a(m)) = \frac{1}{\phi(i) - a(m) - 1} \phi(i) - a(m) + 1,
\]

which should be compared with \(2^M\). We get that the number of codewords for Fib\(m\) will be larger if

\[
\left(\frac{\phi^8(a(m))}{2^M}\right)^M > \frac{(\phi(i) - 1)\phi(i) - a(m)}{a(m)}.
\]

(2)

Nevertheless for \(m = 2\), the constant \(\phi^8/2^7 = 0.367\) is <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

\[
M > \frac{(m - 1) \log_2 \phi(m) + \log_2 (\phi(i) - 1) - \log_2 a(m)}{8 \log_2 \phi(m) - 7},
\]

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...
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 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.

Evident, as because of the possibility of using a larger set of 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 \( 1 \leq i \leq N \), where \( H_n = \sum_{j=1}^{n} (1/j) \) is the \( n \)th harmonic number.\(^1\) It is well known that \( H_n \approx \ln n - \gamma \), 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\( m \), \( m \geq 2 \). 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 \( k \geq m \), 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 obtain that

\[ M_E(N) \approx \frac{\ln N}{\ln 128} \]

and

\[ M_m(N) \approx \frac{\ln(\phi_m - 1) + \ln N - \ln a(m)}{\ln \phi_m} \]

Denoting the \( i \)th codeword by \( C_i \), the average length of a codeword using a Zipf distribution is then

\[
\sum_{i=1}^{N} \frac{1}{i H_N} |C_i| \approx \frac{M_E(N)}{H_N} \sum_{k=1}^{8k} \frac{1}{L_E(k-1)+1} \quad \text{for ETDC}
\]

\[
\approx \frac{1}{H_N} \sum_{k=1}^{M_m(N)} (k + m - 1) \frac{L_m(k)}{L_m(k-1)+1} \quad \text{for Fib} m.
\]

However,

\[
\sum_{L(k-1)+1}^{L(k)} \frac{1}{i} = H_{L(k)} - H_{L(k-1)}
\]

\[
\approx \ln L(k) - \gamma - (\ln L(k-1) - \gamma)
\]

\[
= \ln \frac{L(k)}{L(k-1)}.
\]

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

\[\text{FIGURE 1. Codeword lengths.}\]

\[\text{TABLE 3. Average codeword lengths and excesses over the entropy.}\]

<table>
<thead>
<tr>
<th>Language</th>
<th>Size words</th>
<th>Entropy bits</th>
<th>Huffman</th>
<th>Fib2</th>
<th>Fib3</th>
<th>Fib4</th>
<th>Fib5</th>
<th>Fib6</th>
<th>ETDC</th>
<th>(s, c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>289101</td>
<td>11.173</td>
<td>0.26</td>
<td>10.63</td>
<td>5.35</td>
<td>10.72</td>
<td>18.51</td>
<td>27.01</td>
<td>16.09</td>
<td>(189,67)</td>
</tr>
<tr>
<td>French</td>
<td>439191</td>
<td>10.442</td>
<td>0.29</td>
<td>8.31</td>
<td>4.77</td>
<td>11.09</td>
<td>19.61</td>
<td>28.71</td>
<td>14.82</td>
<td>(185,71)</td>
</tr>
<tr>
<td>Hebrew</td>
<td>296933</td>
<td>13.003</td>
<td>0.23</td>
<td>15.35</td>
<td>6.82</td>
<td>10.44</td>
<td>16.74</td>
<td>23.82</td>
<td>15.20</td>
<td>(176,80)</td>
</tr>
<tr>
<td>Artificial</td>
<td>1000000</td>
<td>13.378</td>
<td>0.20</td>
<td>12.42</td>
<td>3.68</td>
<td>7.04</td>
<td>13.02</td>
<td>19.90</td>
<td>14.07</td>
<td>(171,85)</td>
</tr>
</tbody>
</table>
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

$$\frac{4 \ln 128}{\ln N - \gamma} M_E(N)(M_E(N) + 1)$$

for ETDC,

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

$$\frac{2(\ln N - \gamma)}{\ln \phi(m)} M_{m}(N)(M_{m}(N) + 2m - 1).$$

Figure 2 plots these average values for $N$ up to $10^8$ for ETDC (the bold line) and Fib$m$, with $2 \leq m \leq 6$. 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 $m \leq 5$ and up to alphabet sizes of $10^6$. The additional overhead may reach 33% for alphabets as small as 1000 elements, but even for $N = 1000000$, the overheads of ETDC relative to Fib2, Fib3 and Fib4 are 1, 10 and 7%, respectively.

For a comparison of real data and different languages, the following sets were used, besides 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 17th–20th centuries [19], and for Hebrew, the data was a part of the Responsa Retrieval Project, 100 MB 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 relative 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 [14,15,21], 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 [22], 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, and so the naive approach to decoding, involving many bit manipulations, will be more costly than 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 $T_1T_2T_3\cdots$, and an array $\text{Start}_m[j]$ gives the index of the first element of length $j$ in the Fib$m$ code. Thus $\text{Start}_2[] = 1, 2, 3, 5, 8, \ldots$, $\text{Start}_3[] = 1, 2, 3, 5, 9, 16, \ldots$, $\text{Start}_4[] = 1, 2, 3, 5, 9, 17, 32, \ldots$, 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...
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 Fibn 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 [23] and has been reinvented several times, e.g. [24].

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 $\Lambda$ 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, $\text{Tab}[i, P]$, 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 $\text{Tab}[i, P] = (S, R)$. 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 notation; the decoding tables are labeled by the corresponding prefixes $P$.

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

$$R \leftarrow \Lambda$$

for $i \leftarrow 1$ to length of text in bytes

(output, $R$) $\leftarrow$ Tab[Text[i], $R$]

As an example, consider the input string displayed in Fig. 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 of 227, and therefore 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 $C_3C_4$ 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 has led to various attempts to reduce the number of tables [21,23]. There is one table for each possible prefix of a codeword, and 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 Fibn 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 $\ell$ be the length of the longest of the $N$ codewords, that is, using equation (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
The partial decoding approach is thus not feasible for the large values of \( N \) and \( k \). For example, for \( m = 3 \) and \( N = 4 \), the codewords are \( 111, 0111, 00111 \) and \( 10111 \), and the set of different prefixes includes at least the strings \( 001, 011, 0101, 0011, 101, 1010 \). 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 300 MB.

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 the \( i \)th byte of the text to be decoded, schematically represented in Fig. 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 1s. The second zone consists of zero or more complete codewords, depicted in grey in Fig. 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. In contrast 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 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 \( i > 1 \), \( C_i = d_1 d_2 \cdots d_{i-1}0111 \), where \( rel\_index = \sum_{j=1}^{i} d_j F_j^{(3)} \) 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 \( i = rel\_index + \text{Start}[s+4] \), where \( \text{Start}[i] \) is the index of the

---

**TABLE 4.** Partial decoding tables.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>( \Lambda )</th>
<th>00011</th>
<th>01</th>
<th>100010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>S</td>
<td>R</td>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>220</td>
<td>11011100</td>
<td>C8</td>
<td>00</td>
<td>C5C4</td>
<td>00</td>
</tr>
<tr>
<td>221</td>
<td>11011101</td>
<td>C8</td>
<td>01</td>
<td>C5C4</td>
<td>01</td>
</tr>
<tr>
<td>222</td>
<td>11011110</td>
<td>C8</td>
<td>10</td>
<td>C5C4</td>
<td>10</td>
</tr>
<tr>
<td>223</td>
<td>11011111</td>
<td>C8</td>
<td>11</td>
<td>C5C4</td>
<td>11</td>
</tr>
<tr>
<td>224</td>
<td>11100000</td>
<td>C1</td>
<td>0000</td>
<td>C5</td>
<td>110000</td>
</tr>
<tr>
<td>225</td>
<td>11100001</td>
<td>C1</td>
<td>0001</td>
<td>C5</td>
<td>110001</td>
</tr>
<tr>
<td>226</td>
<td>11100010</td>
<td>C1</td>
<td>0010</td>
<td>C5</td>
<td>110010</td>
</tr>
<tr>
<td>227</td>
<td>11100011</td>
<td>C1</td>
<td>0011</td>
<td>C5</td>
<td>110011</td>
</tr>
</tbody>
</table>
first codeword of length \( \ell \), as mentioned above in the description of the naive decoding algorithm.

Suppose that the codeword \( C_i = d_1d_2 \cdots d_0111 \) is split after the \( t \)th bit into two parts that are treated separately: \( D = d_1d_2 \cdots d_t \) and \( E = d_{t+1} \cdots d_0111 \). 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 \( EV = \sum_{j=t+1}^{p} d_j F_j^{(3)} \). For example, if \( C_i = 010110110111 \) and \( t = 5 \), we get \( D = 01011 \) and \( E = 10110111 \), 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 \( EV \). However, here we can use the fact that \( F_{j+1}^{(3)} = \phi_j^{(3)} \times F_j^{(3)} \).

One would thus expect the requested value to be \( V = DV + EV \times \phi_j^{(3)} \), rounded to the nearest integer. However, in our example, \( DV + EV \times \phi_j^{(3)} = 271.60 \), and hence 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 \( \phi_j \) to the power \( j \), multiplying by \( a(3j) \) and then rounding, this property is not additive: when substituting, in our approximation, \( F_j^{(3)} \) by \( a(3j)\phi_j^{(3)} \), the cumulative error for several such substitutions is not necessarily \(<1/2 \), 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 \( EV = \sum_{j=t+1}^{p} d_j F_j^{(3)} \), we define \( EV = \sum_{j=t+1}^{p} d_j \phi_j^{(3)} \), 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 \( \phi_j^{(3)} \). For our example, one gets thus \( DV = 18.958 \) and \( EV = 12.063 \), so that \( DV + EV \times \phi_j^{(3)} = 272.881 \); 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:

1. **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 \( PL = k \);
2. **PV**: the approximate value of this prefix, using \( \phi_j^{(3)} \);
3. **S**: a sequence of zero or more full codewords that could be decoded from the byte;
4. **SL**: the length of the suffix of the byte that has not been decoded;
5. **SV**: the approximate value of this suffix, as above;
6. **Ev**: the value of the entire byte, \( 0 \leq p < m \).

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 five 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 Fig. 5. The procedure uses a precomputed array \( \Phi \), defined by \( \Phi[i] = \phi_i^{(3)} \); \( oldSV \) and \( oldSL \) 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 \( p \geq 1 \), 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, and hence 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 \( a(3)\phi_j^{(3)}oldSL \), and therefore if one defines \( PV = -a(3) \), the definition of \( index \) in line 8 also applies in this case. For \( p = 2 \), the amount to be

```plaintext
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 \( C_{\text{index}} \), S
10 (oldSV, oldSL) ← (SV, SL)
```

**FIGURE 5.** Revised decoding procedure with partial decoding tables.
subtracted is
\[ a_{(3)}\phi_{(3)}^{\text{oldSL} - 1} + a_{(3)}\phi_{(3)}^{\text{oldSL}} = a_{(3)} \left( 1 + \frac{1}{\phi_{(3)}} \right) \phi_{(3)}^{\text{oldSL}} = 1.4738 \phi_{(3)}^{\text{oldSL}}, \]

and hence 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, and hence 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 may rather 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, and so for $m = 3$ and $k = 8$, the total size is $2^8 \times 3 \times 12. < 10$K. 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 $2^{16} \times 3 \times 13. < 2.5$ MB, which is often acceptable.

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

<table>
<thead>
<tr>
<th>Index</th>
<th>$p = 0$</th>
<th>$p = 1$</th>
<th>$p = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PL</td>
<td>PV</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>80.9989</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>124</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>125</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>126</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>127</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>129</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>130</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>131</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>132</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>133</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>134</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>135</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
<tr>
<td>136</td>
<td>0</td>
<td>1</td>
<td>1.1374</td>
</tr>
</tbody>
</table>

...
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. However, 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 all values in the tables. Thus \( \phi(3) \), \( \phi(5) \), \( \phi(8) \), and \( \phi(13) \) must be replaced by \( \phi(3) \), \( \phi(5) \), \( \phi(8) \), and \( \phi(13) \) respectively, by numbers \( p \) of the form \( p = 2^k \).

3.3.3. Eliminating multiplications

One may still object that the use of 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 \( \Phi(p) \) in lines 5 and 8 by pre-calculated values of shifted Fibonacci numbers. One thus prepares a two-dimensional table \( \text{Fib3} \) and \( \text{shift} \), \( \text{index} \) running over all the possible PV values, and \( \text{shift} \) 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 bit string \( x_1 \cdots x_k \) represents the integer \( \sum_{i=1}^{k} x_i F_{i}^{(3)} \).

**TABLE 6. Influence of precision on the correctness.**

<table>
<thead>
<tr>
<th>Precision</th>
<th>( a(3) )</th>
<th>( \phi(3) )</th>
<th>Index of first wrong codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.61841999</td>
<td>1.8392868</td>
<td>189473</td>
</tr>
<tr>
<td>64/32</td>
<td>0.6184199223193926</td>
<td>1.8392867552141611</td>
<td>2097155</td>
</tr>
<tr>
<td>64</td>
<td>0.6184199223193926</td>
<td>1.8392867552141611</td>
<td>–</td>
</tr>
</tbody>
</table>

The Computer Journal, 2009

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. However, 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 all values in the tables. Thus \( \phi(3) \), \( \phi(5) \), \( \phi(8) \), and \( \phi(13) \) must be replaced by \( \phi(3) \), \( \phi(5) \), \( \phi(8) \), and \( \phi(13) \) respectively, by numbers \( p \) of the form \( p = 2^k \).

3.3.3. Eliminating multiplications

One may still object that the use of 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 \( \Phi(p) \) in lines 5 and 8 by pre-calculated values of shifted Fibonacci numbers. One thus prepares a two-dimensional table \( \text{Fib3} \) and \( \text{shift} \), \( \text{index} \) running over all the possible PV values, and \( \text{shift} \) 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 bit string \( x_1 \cdots x_k \) represents the integer \( \sum_{i=1}^{k} x_i F_{i}^{(3)} \). \( \text{Fib3} \) will contain the corresponding value obtained by shifting all the bits by \( \text{shift} \) bits to the right, i.e.

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

Table 7 displays some sample lines and columns of the \( \text{Fib3} \) table. The last two lines are a special case, to be explained below.

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

\[
\begin{align*}
5 & \quad \text{oldSV} \leftarrow \text{oldSV} + \text{Fib3}[PV, \text{oldSL}] \\
8 & \quad \text{index} \leftarrow \text{oldSV} + \text{Fib3}[PV, \text{oldSL}] + \text{Start}[3 + \text{oldSL} + \text{PL}]
\end{align*}
\]

Note that all the values are integers now; therefore no rounding is necessary and there are no multiplications. The tables \( \text{Tab}[j] \) 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 \( F_{\text{oldSL}}^{(3)} \) to \( \text{oldSV} \), 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 \( F_{\text{oldSL}−1}^{(3)} + F_{\text{oldSL}}^{(3)} \).

As above, the correct value of \( \text{PL} \) in these cases is \( −p \). The correction factors will be accommodated in the \( \text{Fib3} \) 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 1011011011011, representing $M = 19511$. The new entries are defined as

$$Fib3[M + 1, s] = -F(3)^s$$
$$Fib3[M + 2, s] = -F(3)^s - F(3)^{s-1} - F(3)^{s-2}.$$

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.

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 $N = 10^4$, 105 and 106, 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 Fig. 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 6 GB 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 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. However, this would disqualify non-sequential searches, which can be much more efficient, such as the Boyer–Moore (BM) algorithm [26]. Thus, 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 <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 only at the end of each codeword. However, for all the four codes mentioned, if a given codeword \(E(x)\) is located in the compressed text \(E(T)\), 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 that 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 1000100011111011001111, 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, and therefore to find \(E(x)\), one can look for 0111\(E(x)\) in 0111\(E(T)\). This would come with a penalty of increasing the lengths of certain codewords, yielding, an average length of 9.52 and 11.27 bits respectively, for the Bible and the French texts used above, an increase of <2%. This could be tolerable as the files would still be shorter by 7–9% than the corresponding optimal SCDC variants.

As a second option, when better compression is critical, the search procedure in the Fib3 encoded text can be similarly amended to the additional check of the byte preceding the match in tagged Huffman or SCDC. To locate \(E(x)\), one searches for the pattern 111\(E(x)\), 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 a potentially infinite sequence of checks, but practically,

<table>
<thead>
<tr>
<th>Database</th>
<th>Total no of words</th>
<th>No. of different words</th>
<th>Compressed size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fib3</td>
<td>SCDC</td>
</tr>
<tr>
<td>Bible KJV</td>
<td>611793</td>
<td>11378</td>
<td>9.34</td>
<td>(225,33)</td>
</tr>
<tr>
<td>French</td>
<td>1176192</td>
<td>75192</td>
<td>11.12</td>
<td>(188,68)</td>
</tr>
</tbody>
</table>

TABLE 10. Empirical comparison of Fib3 and SCDC decoding.
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 \( \text{the} \). 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 consecutive 111s. The phrase \( \text{the the} \) 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 had 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.

Similar 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 fuse 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Decoded text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib3</td>
<td>And the earth was without form and — darkness was upon the face of the deep. And the Spirit of God moved upon...</td>
</tr>
<tr>
<td>Huff</td>
<td>And the earth was without form and void and therefore in the Tamar that and the was we shall Spirit of God moved...</td>
</tr>
<tr>
<td>SCDC</td>
<td>And the earth was without mouth whom to hands with to sword for but at and eat And and voice took his and...</td>
</tr>
</tbody>
</table>

### Table 11. Empirical comparison of compressed search (in milliseconds).

<table>
<thead>
<tr>
<th>Database</th>
<th>SCDC</th>
<th>Fib3 (Byte)</th>
<th>Fib3 (Bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bible KJV</td>
<td>0.88</td>
<td>2.48</td>
<td>81.7</td>
</tr>
<tr>
<td>French</td>
<td>0.67</td>
<td>2.04</td>
<td>65.5</td>
</tr>
</tbody>
</table>

### Table 12. Illustration of robustness: decoded text after having deleted the bit at position number \(i\).
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 0101000111 (earth) turned into 0101000111 (before); for \( i = 160 \), the deleted bit was in the separating 111 string, and hence the two codewords 110100110110111 (void) and 0111 (and) fused to 1101001101101111, which represents the index number 12627; there are, however, only 11378 elements in the alphabet, and therefore 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; thus 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, ETDC and SCDC. 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


