Received 18 September 2022, accepted 10 October 2022, date of publication 14 October 2022, date of current version 27 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3214820

RESEARCH ARTICLE

Generalization of Fibonacci Codes to the Non-Binary Case

SHMUEL T. KLEIN[1], **TAMAR C. SEREBRO**[2], **AND DANA SHAPIRA**[2]
[1]Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[2]Department of Computer Science, Ariel University, Ari'el 40700, Israel
Corresponding author: Dana Shapira (shapird@g.ariel.ac.il)

ABSTRACT Motivated by expected technological developments in which the basic unit of storage might possibly be d-ary elements, with $d > 2$, and not just bits, we extend the traditional Fibonacci code to non-binary codes of higher order, and prove their theoretical properties, as follows: (1) these codes are *fixed in advance*, and therefore do not need to be generated for each new probability distribution, yielding very simple and fast encoding and decoding procedures; (2) the codes are *prefix-free*: no codeword in the code is the prefix of any other codeword; (3) the codes are *complete*: if one adjoins any other d-ary string as an additional codeword, the obtained extended set of codewords is not uniquely decipherable anymore, and is therefore not useful as it might lead to ambiguities; and (4) the codes provide *robustness* against decoding errors: the number of lost codewords in case of an error will be limited. An error is defined as a d-ary digit changing its value, or an insertion of an extraneous d-ary digit, or an erroneous deletion of a d-ary digit. We provide experimental results on the compression performance, illustrating that the compression efficiency of non-binary Fibonacci codes is very close to the savings achieved by the corresponding non-binary Huffman coding of the same order, while providing simplicity and robustness.

INDEX TERMS Data compression, Fibonacci codes, Huffman coding.

I. INTRODUCTION

The famous Fibonacci series defined by

$$F_0 = 0, \quad F_1 = 1$$

and

$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2,$$

has contributed to several applications in Data Compression. Quite a few encodings have been suggested that are based on the properties of the Fibonacci sequence, and can be used as alternatives to fixed length codes. The Fibonacci encoding is built on the following simple procedure.

A first step is to decide about the *alphabet* to be encoded. This is often just a standard set of characters, such as ascii, but can also be a much larger set of variable lengths strings, which yields much better compression. A popular choice is the set of all different *words* in a textual database as in [28]. When we refer to an alphabet, it should thus be understood in this much broader sense, and one should bear in mind that its size may be in the millions in actual applications, such as large Information Retrieval Systems [3].

Once the alphabet is given, the text has to be parsed into a sequence of elements of this alphabet, and statistics are gathered to derive the corresponding probability distribution. While Huffman's optimal algorithm then constructs a prefix tree with minimal average depth of its leaves, corresponding to the codewords that are assigned to the alphabet elements, the procedure for Fibonacci codes, and other static codes such as [3] or [8], is much simpler: all it needs is to order the elements by non-increasing weights, and then assign to them, in order, the codewords for the integers $1, 2, 3, \ldots$. The encoded file thus consists just of a sequence of integers, which can efficiently be decoded with the help of a simple mapping. In certain applications, the sorting step can be omitted, see [21].

The associate editor coordinating the review of this manuscript and approving it for publication was Cesar Vargas-Rosales.

The basic Fibonacci code is a universal [8] variable length encoding of the integers based on the Fibonacci sequence rather than on the sequence of powers of 2, see [12] and [17]. Formally, the Fibonacci code is a binary representation of the integers based on the numeration system composed of the Fibonacci sequence: any integer x can be written as $x = \sum_{i\geq 2} f_i F_i$, with $f_i \in \{0, 1\}$.

The representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number. If indeed this algorithm is used, it implies that the Fibonacci representation of any integer will never include consecutive Fibonacci numbers, in other terms, the binary encodings of the integers are bitstrings that do not contain adjacent 1's. The addition of a single 1-bit to each integer representation is therefore enough to turn this sequence into a uniquely decipherable variable length code.

The standard binary encoding considers the basis elements $1, 2, 4, \ldots$ from right to left, so that for example, the number $28 = 16 + 8 + 4$ is represented by the string 11100. When the Fibonacci sequence is used instead of the powers of 2, the equivalent would be $28 = 21 + 5 + 2$ yielding the string 1001010, but it is convenient to reverse the string to 0101001 and consider the basis elements from left to right. The separating 1-bit can then be appended at the right end, giving here 01010011. Since after reversing, the rightmost bit will be leading and thus always be 1, the resulting set of codewords will be a prefix code. A *prefix (suffix) code*, often more accurately called *prefix-free (suffix-free)*, is a set C of codewords satisfying the constraint that no codeword in C is the proper prefix (suffix) of any other codeword in C. A major property of prefix and suffix codes is that they are uniquely decipherable, and that this can be done instantaneously without delay [29].

Actually, the property of non-adjacency of 1-bits can be used as equivalent definition of the Fibonacci code as the set of codewords which consists of all the binary strings in which the substring 11 appears exactly once, at the right end of the string. This yields the prefix code $\mathcal{E}_{fib} = \{11, 011, 0011, 1011, 00011, 10011, 01011, 000011, 100011, 010011, 001011, 101011, 0000011, \ldots\}$.

Interest in the Fibonacci sequence has meanwhile shifted also to other adjacent areas, such as *compressed matching*, first mentioned in [1], in which a pattern is to be located in a text, which is assumed to be given in some compressed form. The challenge is then to handle the search of the *compressed* pattern in the *compressed* text, if possible, rather than to decompress the text and then search within it. The advantage of using Fibonacci codes in this context is the 11 separator, that acts as a border between adjacent codewords. Compressed matching is not limited to text files only, in [20] and [22], the Fibonacci code was adapted to tools in image compression.

It should be emphasized that from the compression point of view, the static Fibonacci codes are obviously not competitive with the optimal Huffman codes. They are a noteworthy alternative only if other criteria are taken into account:

1) Their set is fixed in advance and need not to be generated for each new probability distribution;
2) encoding and decoding procedures are very simple and thus fast;
3) they provide robustness against decoding errors, such as bit-flips, insertions of extraneous bits or erroneous deletions of bits: in any case, the damage caused by such errors will be limited.

Moreover, for large enough alphabets, the loss in compression efficiency versus Huffman codes may also be quite small. In any case, the replacement of the optimal Huffman codes by codeword sets that are fixed in advance and therefore necessarily sub-optimal, is a well accepted practice to gain improvements for other criteria, for example in [5], which introduce directly accessible codes (DACs) by integrating rank data structures into variable lengths codes, or in the dense codes introduced in [4] and [3]. We explore in this paper the extension of the binary Fibonacci code to d-ary codes, with $d \geq 2$. A one page abstract of this work has appeared in the Proceedings of the Data Compression Conference (DCC'20) [18]. This might be motivated by future technological developments in which the basic unit of storage will not be just a 2-valued bit, but possibly an element that is able to distinguish between d different values, see [6]. For $d = 3$, such *ternary* digits are often called *trits*, and a ternary computer based on 3-valued ternary logic has been built in the Soviet Union already in 1958, see [31].

Codes of order d are related to d-ary trees and there are several motivations for using such trees with $d > 2$. A d-ary tree is of height $\lceil \log_d |\Sigma| \rceil$, which might improve the processing time for larger d, e.g., in higher order *Wavelet trees*, where instead of storing binary bitmaps in every internal node as in the binary Wavelet trees introduced by [14], one rather stores sequences over the alphabet $\{1, \ldots, d\}$. Ferragina et al. [9] show how to handle *rank* and *select* of such sequences in constant time. This improves both time and space complexities. For more details on Wavelet trees, see the survey of [30].

A Fibonacci Wavelet tree, in which pruning was applied for additional savings, was defined in [23]. This data structure has then been generalized to higher-order of another kind, namely by the use of higher order Fibonacci Codes [12], where each element of the underlying sequence is the sum of the d preceding ones, for $d \geq 2$. The corresponding binary code has the property that there is no occurrence of a string of d consecutive 1s. The special case $d = 2$ is the original Fibonacci code.

Another field of application for non-binary codes is *Write-Once Memories* (WOM), a popular example of which is *flash* memory, as can be found now in many electronic devices. In fact, higher order WOM codes were already suggested in the work of [10], and is today a rich research field on its own [33]. In a generalized model of flash memory, each cell can store d possible linearly-ordered values, with the

TABLE 1. First 30 codewords of $(m+1)$-ary Fibonacci codes.

	$m=1$	$m=2$	$m=3$	$m=4$		$m=1$	$m=2$	$m=3$	$m=4$
1	11	12	13	14	16	0010011	2022	3013	134
2	011	22	23	24	17	1010011	00012	0113	234
3	0011	012	33	34	18	0001011	10012	1113	334
4	1011	112	013	44	19	1001011	20012	2113	434
5	00011	212	113	014	20	0101011	01012	3113	044
6	10011	022	213	114	21	00000011	11012	0213	0014
7	01011	0012	313	214	22	10000011	21012	1213	1014
8	000011	1012	023	314	23	01000011	02012	2213	2014
9	100011	2012	123	414	24	00100011	00112	3213	3014
10	010011	0112	223	024	25	10100011	10112	0313	4014
11	001011	1112	323	124	26	00010011	20112	0023	0114
12	101011	2112	033	224	27	10010011	01112	1023	1114
13	0000011	0212	0013	324	28	01010011	11112	2023	2114
14	1000011	0022	1013	424	29	00001011	21112	3023	3114
15	0100011	1022	2013	034	30	10001011	02112	0123	4114

constraint that rewrites can only *increase* a cell's value [7]. In the binary case, each cell can be changed from level zero to level one, but not vice versa. As an illustration, consider a punch card with holes of three different sizes, small, medium and large. Every hole can only become larger. There is no option of decreasing the size of a hole, and if such action is needed, the card must be replaced.

Codes based on generalizations of the Fibonacci sequence have already been mentioned in [16]: a (d, k)-limited binary sequence, also called *runlength limited* (RLL), contains at least d and at most k zeros between any two 1-bits. Therefore, the codewords of $\mathcal{E}_{fib}$, without their ending 1-bits, are the set of $(1, \infty)$-limited sequences.

These codes are then extended to *higher level*, in which M-ary, and not just binary, digits may be used. Several properties of such M-ary RLL codes are reported in [27] and [26].

To avoid a confusion between the different generalizations (higher order or level could mean non-binary, but also that more than two elements are added in the recursive definition), we shall refer in this paper explicitly to non-binary codes (even for the special case $d = 2$). The paper is constructed as follows: Section 2 presents the proposed non-binary codes and some of their properties and Section 3 provides experimental results about their compression performance.

II. NON BINARY FIBONACCI CODES

Consider the following generalization of the standard Fibonacci sequence, depending on a parameter $m \geq 1$. Define the family of sequences

$$R^{(m)}_{-1} = 1, \quad R^{(m)}_0 = 1,$$
$$R^{(m)}_i = mR^{(m)}_{i-1} + R^{(m)}_{i-2} \qquad \text{for } i > 0. \tag{1}$$

For $m = 1$, this is the standard Fibonacci sequence F_i. The numeration system based of the sequence $\mathcal{R}^{(m)} = \{R^{(m)}_0, R^{(m)}_1, R^{(m)}_2, \ldots\}$ is a $d = (m + 1)$-ary system,[1] i.e., any integer L can be uniquely represented as $L = \sum_i a_i R^{(m)}_i$, where the coefficients a_i of the basis elements are not just binary, but belong to a larger set $0 \leq a_i \leq m$. The additional property, generalizing the non-adjacency of 1-bits of the Fibonacci encoding, is that if a_{i+1} reaches its maximal permitted value m, then the digit a_i just preceding it, if there is such a digit, has to be zero [11]. Note that the generalization of the Fibonacci sequence given in eq. (1) is different from the extension to higher level mentioned in part 4.7 of [16]; in particular, in the defining recurrence in eq. (4.73) of [16], the level parameter m multiplies the second term, and not the first one as in our definition given in eq. (1). Therefore the additional property that

$$a_{i+1} = m \quad \rightarrow \quad a_i = 0$$

does not hold for this other generalization.

As example, the elements of the ternary numeration system $\mathcal{R}^{(2)}$ are $\{1, 3, 7, 17, 41, \ldots\}$, and the sequence of the first codewords, representing the integers 1 to 20 according to $\mathcal{R}^{(2)}$, is

1, 2,
10, 11, 12, 20,
100, 101, 102, 110, 111, 112, 120, 200, 201, 202,
1000, 1001, 1002, 1010, ...

Turning this representation into a useful code can be done by the following steps:

1) exploit the fact that integers are represented without leading zeros, so that the leftmost digit is one of $\{1, 2, \ldots, m\}$; we can thus prefix the representation of any integer by the digit m, which will act as a comma between codewords, because within a codeword, the digit m must be followed (in a left to right scan) by 0.
2) reverse all the codewords.

Step 2 turns the resulting set into a prefix code, which is instantaneously decodable, as shown in the following lemma.

[1] The reason for switching notation from d used so far to m is to avoid confusion, since the new numeration system is $(m + 1)$-ary.

Lemma 1: For all $m \geq 1$, the infinite code obtained by the m-ary extension of the Fibonacci code given above is prefix-free.

Proof: Note that every codeword w terminates on its right end by a pair of digits xm, with $x \neq 0$. It follows that w can not be the prefix of any longer codeword, because such xm does not appear anywhere else in any of the codewords. ■

The first thirty codewords of these sets, for $1 \leq m \leq 4$, are shown in Table 1.

An alternative, equivalent, definition of this code for $m = 2$ is the sequence of all ternary strings, each terminating with a rightmost trit equal to 2, and with the constraint on all the other trit positions, that if the value of the trit is 2, then the trit is preceded by a trit with value 0, as follows from [11]. Obviously, the constraint does not apply to the leftmost position. To clarify this alternative definition, consider, as example, the representation of the integer 2976 in the ternary system $\mathcal{R}^{(2)}$:

$$2976 = 2 \times 1 + 1 \times 7 + 2 \times 41 + 1 \times 99 + 2 \times 1393.$$

The corresponding ternary string is therefore

$$\underline{2}\ 0\ 1\ 0\ \boxed{2}\ 1\ 0\ 0\ \boxed{2}\ \mathbf{2},$$

in which three different kinds of 2-trits can be seen: the rightmost, terminating 2 is boldfaced, the leftmost, underlined 2 is not subject to any constraint, and the other, boxed, 2-trits, are preceded by 0-trits.

Note that because all the codewords have been reversed, they are not any more in lexicographic order when arranged by increasing values they represent. For example for $m = 2$, the string 212 represents the integer 5, while 022, which is lexicographically smaller, represents 6. Nevertheless, given the codeword $A = a_1a_2 \cdots a_t m$, the order shown in Table 1 enables the calculation of its index $I(A) = \sum_{i=1}^{t} a_i R_i^{(m)}$, so this order is preferable to rearranging the codewords into a different order, for which there is no connection between a codeword and its index.

A. NUMBER OF CODEWORDS OF A GIVEN LENGTH

To evaluate the number of codewords of a given length for the generalized non-binary Fibonacci codes, note that any integer j in the range $R_{i-2}^{(m)} \leq j < R_{i-1}^{(m)}$, requires i digits for its encoding, for every $m \geq 1$ and $i \geq 2$. We thus need a good approximation of the elements of the sequence $R_i^{(m)}$ to know the number of digits necessary to encode an integer i. For the standard binary encoding based on powers of 2, this is just $\log_2 i$. For the basic binary Fibonacci encoding, based on $F_i \simeq \frac{1}{\sqrt{5}}\phi^i$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618$ is the golden ratio, the length of the codeword indexed i is about $\log_\phi i = 1.4404 \log_2 i$ bits.

To evaluate the values for the sequence $R_i^{(m)}$, consider the corresponding homogeneous recurrence relation

$$R_i^{(m)} - mR_{i-1}^{(m)} - R_{i-2}^{(m)} = 0.$$

The two roots of the corresponding second degree equation $x^2 - mx - 1 = 0$ are

$$x_{0,1} = \frac{m \pm \sqrt{m^2+4}}{2}.$$

One can write the ith term of the sequence $R_i^{(m)}$ as a linear combination of the ith powers of the two roots x_0 and x_1, that is, there exist two constants a_0 and a_1 such that for all $i \geq 0$,

$$R_i^{(m)} = a_0\, {x_0}^i + a_1\, {x_1}^i.$$

Using the initial values $R_0^{(m)} = 1$ and $R_1^{(m)} = m+1$, we obtain that

$$a_{0,1} = \frac{\sqrt{m^2+4}\ \pm\ (m+2)}{2\sqrt{m^2+4}}.$$

Table 2 presents the values for a_0, a_1, x_0 and x_1 for $m \leq 3$.

Only one of the roots is larger than 1. The other root is negative and larger than -1. Thus, when representing $R_i^{(m)}$ as a linear combination of the i-th powers of x_0 and x_1, $a_0{x_0}^i$ approximates $R_i^{(m)}$, while $a_1{x_1}^i$ vanishes when i increases. Therefore, the number of digits needed to represent a number i in this $(m+1)$-ary representation is of the order of

$$\log_{x_0} i = \left(\frac{1}{\log_{m+1}(x_0)}\right)\log_{m+1} i.$$

The values $\frac{1}{\log_{m+1} x_0} = \log_{x_0}(m+1)$ are given in the sixth column of Table 2. We see that while the length of the codeword for a given value i, relative to the standard binary encoding, is increased by 44% for the binary Fibonacci code ($m = 1$), the number of trits for the ternary code is only about 25% larger than for the corresponding standard ternary encoding, and for the cases $m = 3, 4$, the increase is reduced to 16% and 11%, respectively. The explanation for the last column of Table 2 is given below.

For a given $m \geq 1$, define $D_i^{(m)}$ as the number of codewords of length i, for $i \geq 2$.

Lemma 2: Given $m \geq 1$, the sequence $D_i^{(m)}$ is defined by the recurrence

$$D_2^{(m)} = m \qquad D_3^{(m)} = m^2,$$
$$D_i^{(m)} = mD_{i-1}^{(m)} + D_{i-2}^{(m)} \qquad \text{for } i \geq 4.$$

Proof: The first element of the $D_i^{(m)}$ sequence is $D_2^{(m)} = m$, because the rightmost digit is always m, and to form a string of length 2, the rightmost m may be preceded by any digit in the range $1, \ldots, m$.

To evaluate $D_3^{(m)}$, note that if the rightmost m is preceded by an additional occurrence of the digit m, the first digit must be 0, so for each m, there is only one choice, namely 011, 022, 033, ... If the rightmost m is preceded by any of the other $m-1$ non-zero digits, the first digit can be any of the $m+1$ possibilities in the range $[0, \ldots, m]$, so the total number of codewords of length 3 is

$$1 + (m+1)(m-1) = m^2.$$

TABLE 2. Roots and coefficients for $m \leq 4$.

m	a_0	x_0	a_1	x_1	$\log_{x_0}(m+1)$	max $\lvert\Sigma\rvert$
1	$\frac{3+\sqrt{5}}{2\sqrt{5}}$	$\frac{1+\sqrt{5}}{2}$	$\frac{\sqrt{5}-3}{2\sqrt{5}}$	$\frac{1-\sqrt{5}}{2}$	1.4404	$2.9 \cdot 10^{17}$
2	$\frac{1+\sqrt{2}}{2}$	$1+\sqrt{2}$	$\frac{1-\sqrt{2}}{2}$	$1-\sqrt{2}$	1.2465	$6.9 \cdot 10^{13}$
3	$\frac{\sqrt{13}+5}{2\sqrt{13}}$	$\frac{1}{2}(3+\sqrt{13})$	$\frac{\sqrt{13}-5}{2\sqrt{13}}$	$\frac{1}{2}(3-\sqrt{13})$	1.1603	$2.9 \cdot 10^{17}$
4	$\frac{1}{2}+\frac{3}{\sqrt{5}}$	$2+\sqrt{5}$	$\frac{1}{2}-\frac{3}{\sqrt{5}}$	$2-\sqrt{5}$	1.1149	$1.9 \cdot 10^{13}$

For $i \geq 4$, recall that an integer j in the range $R_{i-2}^{(m)} \leq j < R_{i-1}^{(m)}$, requires i digits for its encoding, so we get that

$$\begin{aligned} D_i^{(m)} &= R_{i-1}^{(m)} - R_{i-2}^{(m)} \\ &= mR_{i-2}^{(m)} + R_{i-3}^{(m)} - mR_{i-3}^{(m)} - R_{i-4}^{(m)} \\ &= m\big(R_{i-2}^{(m)} - R_{i-3}^{(m)}\big) + \big(R_{i-3}^{(m)} - R_{i-4}^{(m)}\big) \\ &= mD_{i-1}^{(m)} + D_{i-2}^{(m)} \end{aligned}$$

■

The sequence $D_i^{(m)}$ follows therefore the same recurrence relation as the sequence $R_i^{(m)}$, only with different boundary conditions, similarly to Fibonacci and Lucas sequences.

B. ENCODING AND DECODING

As for any universal sequence of strings representing the integers, encoding a list of elements is done by concatenating the corresponding codewords. For example, if we wish to encode the list of numbers 7, 2, 16, 10, the output, when using the ternary code for $m = 2$ and the notations as above, would be

$$0\ 0\ 1\ \mathbf{2}\ \underline{2}\ \mathbf{2}\ \underline{2}\ 0\ \boxed{2}\ \mathbf{2}\ 0\ 1\ 1\ \mathbf{2}.$$

The encoding of a single number can be done in constant time by means of an encoding table T, storing at entry i the $(m+1)$-ary Fibonacci representation of the integer i. More precisely, suppose every entry of T consists of 64 bits, which is quite standard for current computers. We allocate the $6 = \log_2 64$ leftmost bits of each entry i to encode the number of $(m+1)$-ary digits in the representation of i; denote this number by ℓ_i. Each of these digits can be stored in $\lceil \log_2(m+1) \rceil$ bits in the bits after ℓ_i. In our example above, the number 2976 has been represented in the ternary Fibonacci code as the sequence 2010210022 of ten trits; the rightmost 2 need not to be explicitly stored in the table, so the leftmost 24 bits of entry number 2976 for encoding the remaining nine trits would be

$$001001\ 10\ 00\ 01\ 00\ 10\ 01\ 00\ 00\ 10 \cdots,$$

where spaces have been inserted for clarity between the digits and to separate ℓ_i from the digits following it. The number of different $(m+1)$-ary strings which can be stored in this way in a 64-bit entry is therefore bounded by

$$(m+1)^{\lfloor (64-6)/\lceil \log_2(m+1) \rceil \rfloor},$$

and these numbers are shown in the last column of Table 2, headed by max $|\Sigma|$. We see that these numbers are larger than we probably ever will need. Even if we reduce the size of a table entry to 32 bits, one may still accommodate alphabets of at least 1.6 million elements for $m < 8$. It therefore follows that for any practical application in the foreseeable future, the following holds:

Lemma 3: Given $m \geq 1$ and any integer i, the time complexity of encoding any integer represented in the m-ary Fibonacci code is $O(1)$.

For decoding, we have to locate the codeword boundaries. This is done in a left to right scan, checking repeatedly whether the defining rule, that every digit m must be preceded by a zero, is fulfilled. If it is, we are in the middle of a codeword and thus continue with the scan; if the rule is broken, we know that a codeword boundary has been reached. This is conveniently summarized by the automata in Figure 1, the left one showing the special case for $m = 2$ and the right one depicting the automaton for general m.

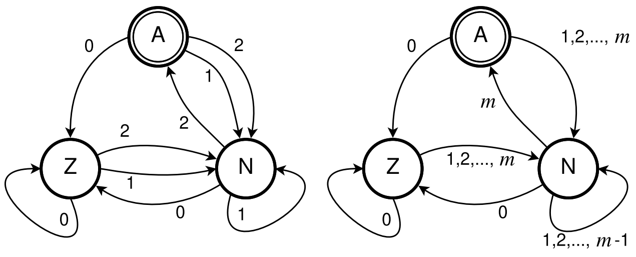


FIGURE 1. Decoding automaton.

The accepting state A is reached at the end of each codeword; this is also the initial state. If a zero is scanned, there is a transition into state Z, and for any other digit (1 or 2 for $m = 2$), the transition is into state N, representing the Non-zero digits. If, while in state N, we see the digit m, we have reached a boundary, and the current codeword consists of the digits scanned since the last visit in state A; a zero digit transfers us back into state Z, and every other digit lets us stay in state N. Running the example input string 00122220220112 through the left hand side automaton correctly parses the string into 0012 22 2022 0112, which can be translated into 7, 2, 16, 10 by single table accesses.

Lemma 4: Given $m \geq 1$ and any integer i, the time complexity of decoding the integer i represented in the m-ary Fibonacci code by means of the automaton of Figure 1 is logarithmic in i.

Proof: The number of steps executed by the automaton is clearly linear in the length of its input, which is the $(m+1)$-ary Fibonacci representation of i. This has been shown above to be of length $O(\log_{m+1} i)$. ∎

C. COMPRESSION PERFORMANCE

Having clarified that the $(m+1)$-ary Fibonacci codes are clearly inferior, from the compression point of view, to the optimal Huffman codes, we still deal in this section with the expected compression performance, to enable a decision of whether the trade-off of improving other features at the cost of reduced compressibility is worthwhile. Suppose then that we are given an alphabet on size n, and a probability distribution $P = \{p_1, p_2, \ldots, p_n\}$ of its elements. For example, one could derive P from pre-processing a given input text T. We further assume that the alphabet is already given sorted by weight, that is, $p_i \geq p_{i+1}$ for $1 \leq i < n$. As explained in the introduction, the encoding procedure is simply assigning the integer i, represented in the $(m+1)$-ary Fibonacci code, to the i-th element.

Let $\ell_i^{(m)}$ denote the length of the $(m+1)$-ary codeword for the integer i. The values of this sequence are constant (the first ones can be derived from Table 1), for example, for $m = 1$, the sequence $\ell^{(1)}$ starts with $2, 3, 4, 4, 5, 5, 5, 6, \ldots$, for $m = 2$, $\ell^{(2)}$ starts with $2, 2, 3, 3, 3, 3, 4, \ldots$, etc. The expected length of a codeword in the entire file is therefore $\sum_{i=1}^{n} \ell_i^{(m)} p_i$.

To compare the codes analytically on some distributions that are more realistic than the uniform one, consider first *Zipf's law* [34], which is believed to well estimate the frequency of the different words in a large natural language corpus, as well as many other natural phenomena. It is defined by the weights

$$p_i = \frac{1}{i H_n} \quad \text{for } 1 \leq i \leq n, \qquad \text{where } H_n = \sum_{j=1}^{n} \frac{1}{j}$$

is the n-th harmonic number, well known to be about $\ln n$. Using $n = 200$, the first few elements of the sequence P are $0.170, 0.085, 0.057, 0.043, \ldots$, and encoding P using $(m+1)$-ary Fibonacci codes yields the average values shown in the line headed Zipf of Table 3. The following line shows the excess, in percent, of using the Fibonacci variants rather than the corresponding optimal $(m+1)$-ary Huffman codes.

As second example, we started from distributions for which Huffman coding is known to yield zero redundancy, so that their average codeword length coincides with the entropy, $-\sum_{i=1}^{n} p_i \log_2 p_i$. In the binary case, such a distribution is called *dyadic*, because all the probabilities are powers of $\frac{1}{2}$. Given a general distribution $P = \{p_1, p_2, \ldots, p_n\}$, consider the lengths $\{l_1, l_2, \ldots, l_n\}$ of the codewords of the corresponding binary Huffman code; one can often use the dyadic distribution $D = \{d_1, d_2, \ldots, d_n\}$, where $d_i = 2^{-l_i}$, as approximation for P, because both distributions yield the same Huffman tree [25].

The distribution corresponding to a ternary Huffman code should then be *triadic*, that is, consisting of negative powers of 3, and in general, a $(m+1)$-adic distribution consists of probabilities of the form $(m+1)^{-l_i}$. We took the lengths $\ell_i^{(m)}$ of the 200 first codewords of the $(m+1)$-ary Fibonacci codes, derived the corresponding $(m+1)$-adic probabilities, and then normalized to get a distribution. The resulting average codeword lengths are in Table 3 in the line headed $(m+1)$-adic, and their excess over the corresponding optimal $(m+1)$-ary Huffman codes are in the next row.

We note that in the binary case, the compression loss incurred by using the Fibonacci variants is quite small, but seems to be increasing with m. Remember, however, that one of the main motivations for using m-ary codes is the possibility of the emergence of new technologies which will enable the use of $d > 2$ values per storage unit, and the examples in Table 3 clearly indicate that higher order methods are preferable.

TABLE 3. **Comparing compression of Fibonacci vs. Huffman coding for $m \leq 4$.**

m	1	2	3	4
Zipf	6.36	4.15	3.48	3.17
Excess over Huff	5.5%	8.3%	14.3%	20.4%
$(m+1)$-adic	4.80	3.73	3.30	3.05
Excess over Huff	2.4%	5.5%	8.7%	12.0%

D. COMPLETENESS

The exact number of codewords of each length, as derived in Lemma 2, can be used to show the completeness of the code. A d-ary code $\mathcal{C}$ is said to be complete, if adjoining to it any d-ary string $s \notin \mathcal{C}$ yields a set of codewords $\mathcal{C} \cup \{s\}$ that is not uniquely decipherable, in other words, the obtained set is not a useful code as it might lead to ambiguities. An equivalent condition for a d-ary prefix code to be complete is that it satisfies the McMillan equality $\sum_{i \geq 1} n_i d^{-i} = 1$, where n_i is the number of codewords of length i.

Lemma 5: For all $m \geq 1$, the infinite code obtained by the m-ary extension of the Fibonacci code given above is complete.

Proof: Recall that for a given $m \geq 1$, the extended Fibonacci code is an $(m+1)$-ary prefix code. Denote then by $A = \sum_{i \geq 2} D_i^{(m)} (m+1)^{-i}$ the corresponding McMillan sum. We show that $A = 1$ by using the explicit values for $i \leq 3$ and applying the recursion for $i \geq 4$, as given in Lemma 2:

$$A = D_2^{(m)} (m+1)^{-2} + D_3^{(m)} (m+1)^{-3} + \sum_{i=4}^{\infty} D_i^{(m)} (m+1)^{-i}$$

$$= \frac{m}{(m+1)^2} + \frac{m^2}{(m+1)^3} + \sum_{i=4}^{\infty} \left(mD_{i-1}^{(m)} + D_{i-2}^{(m)}\right)(m+1)^{-i}$$

$$= \frac{m}{(m+1)^2} + \frac{m^2}{(m+1)^3} + \frac{m}{m+1} \sum_{i=4}^{\infty} D_{i-1}^{(m)}(m+1)^{-(i-1)} \quad (2)$$

$$+ \frac{1}{(m+1)^2} \sum_{i=4}^{\infty} D_{i-2}^{(m)}(m+1)^{-(i-2)}. \quad (3)$$

But the summation in eq. (3) is just A, and the sum in eq. (2) is also A, from which the first term $D_2^{(m)}(m+1)^{-2} = \frac{m}{(m+1)^2}$ is missing. Rearranging the terms we get

$$A \left[1 - \frac{m}{m+1} - \frac{1}{(m+1)^2}\right] = \frac{m}{(m+1)^2} + \frac{m^2}{(m+1)^3} - \frac{m}{m+1} \frac{m}{(m+1)^2}.$$

Both the right hand side of the equation, and the value of the expression in brackets, multiplying A on the left hand side, are equal to $\frac{m}{(m+1)^2}$ and non-zero, implying that $A = 1$. ■

E. ROBUSTNESS

We start by restricting our attention to single digit errors that occur by a digit insertion or deletion, or by the replacement of one of the digits by a different one. As robustness measure, we use the one defined in [12] as the maximal number of affected codewords. Accordingly, fixed length codes are extremely vulnerable, since a single inserted or deleted bit might render the suffix of the text after the error useless. Huffman codes have a tendency to resynchronize, a property that has been exploited for pattern matching in Huffman encoded files [19], but in the worst case, e.g., when the underlying Huffman code is both a prefix and a suffix code, the decoded output may be completely garbled after an error [13]. For binary Fibonacci codes, on the other hand, the number of lost codewords has been shown to be bounded by 3 in [12].

The same bound also applies to the non-binary Fibonacci codes defined herein, and in particular, the robustness measure does not depend on m.

Lemma 6: For all $m \geq 1$, the number of lost codewords in case of a single m-ary digit-error is bounded by 3.

Proof: As a worst case example, consider the sequence `0033 33 033` for $m = 3$, representing the numbers 39, 3, 12. If the penultimate digit of the first codeword is replaced by `0`, the sequence becomes `00033 3033` (spaces are inserted for clarity) that represent the numbers 129, 42, missing the correct decoding of three codewords. Replacing every `3` in this example by m gives an example for general m.

That this is indeed the worst case can be argued as follows. Given is a codeword $A = a_1a_2 \cdots a_{r-2}a_{r-1}m$ of the $(m+1)$-ary code, that is, $a_i = m$ implies $a_{i-1} = 0$ for $1 \leq i < r$ and $a_{r-1} > 0$. We consider the effect of an error according to the following possible three cases:

1) If the error occurs in any of the first $r - 2$ positions of A, only A itself is lost, possibly having it decoded erroneously as two codewords. More precisely, adding, at one of these positions, any of the digits 0 to $m - 1$, or adding a digit m after an occurrence of `0`, causes the loss of A alone, while no other codewords are affected. Adding a digit m after a digit which is not `0` causes the split of A into two parts.

   A deletion of a zero, in case it is followed by m, may cause the split of A into two parts. Otherwise, a deletion of any digit at any position other than the rightmost two of any codeword does only affect A itself, which will be decoded erroneously. Changing a `0` into another digit will again split A into two, in case it is followed by m. Otherwise, A is changed into another codeword, but in both cases, only a single codeword is lost.
2) If an error occurs in the rightmost digit, changing the value m, or inserting a digit other than m at that position, or deleting that digit in case it is not followed (as the first digit of the following codeword) by m, two adjacent codewords, A and the one following it, are interpreted as a single codeword.
3) If an error occurs in a_{r-1}, and this digit is replaced by `0`, then the terminating m of A is not recognized as a separator, so A and the following codeword are merged into one, while the other codewords are not affected. This is also the case if a_{r-1} is deleted and $a_{r-2} = 0$, or if `0` is inserted between a_{r-1} and the rightmost digit m. Otherwise, only the affected codeword A itself is incorrectly decoded. ■

TABLE 4. Information about the used datasets.

File	Size (MB)	$\lvert\Sigma\rvert$ chars	$\lvert\Sigma\rvert$ words
eng	2108	239	888,343
sources	201	230	611,290
ftxt	7.3	131	41,731
ebib	3.5	53	11,377

Generalizing Lemma 6 to the case of $k > 1$ errors, we see that up to $3k$ codewords may be lost in the worst case, when the different errors are far enough from each other. When the multiple errors occur in bursts in a restricted region, some of them may affect the same codewords, so the number of falsely decoded elements may be strictly smaller. Returning to the example above of `0033 33 033` for $m = 3$, suppose now that the four leftmost 3s are erroneously replaced by 1s; in this case, the three codewords would merge into a single one 001111033, representing the integer 8618, but only 3 codewords would be lost.

An immediate consequence of Lemma 5 is that error detection and correction will not be possible for Fibonacci codes, just as for Huffman's and any other complete code. Indeed, completeness also means that *any* $(m+1)$-ary string can be decoded in a $(m+1)$-ary code, so it will not be possible to distinguish between an original and a corrupted one (except for the trivial case of the final digit not being m). So if error detection and correction are important, the Fibonacci codes have to be extended, as usually done, by checksums or error-correcting codes as in [15].

TABLE 5. Average codeword length per character on file *eng*.

m	1	2	3	4	5	6	7	8
Fibonacci Chars	5.02	3.42	2.93	2.71	2.58	2.51	2.45	2.40
Huffman Chars	4.56	2.92	2.32	2.01	1.83	1.71	1.61	1.54
Fibonacci Words	1.49	0.90	0.72	0.63	0.58	0.54	0.52	0.50
Huffman Words	1.38	0.88	0.70	0.60	0.54	0.50	0.47	0.45

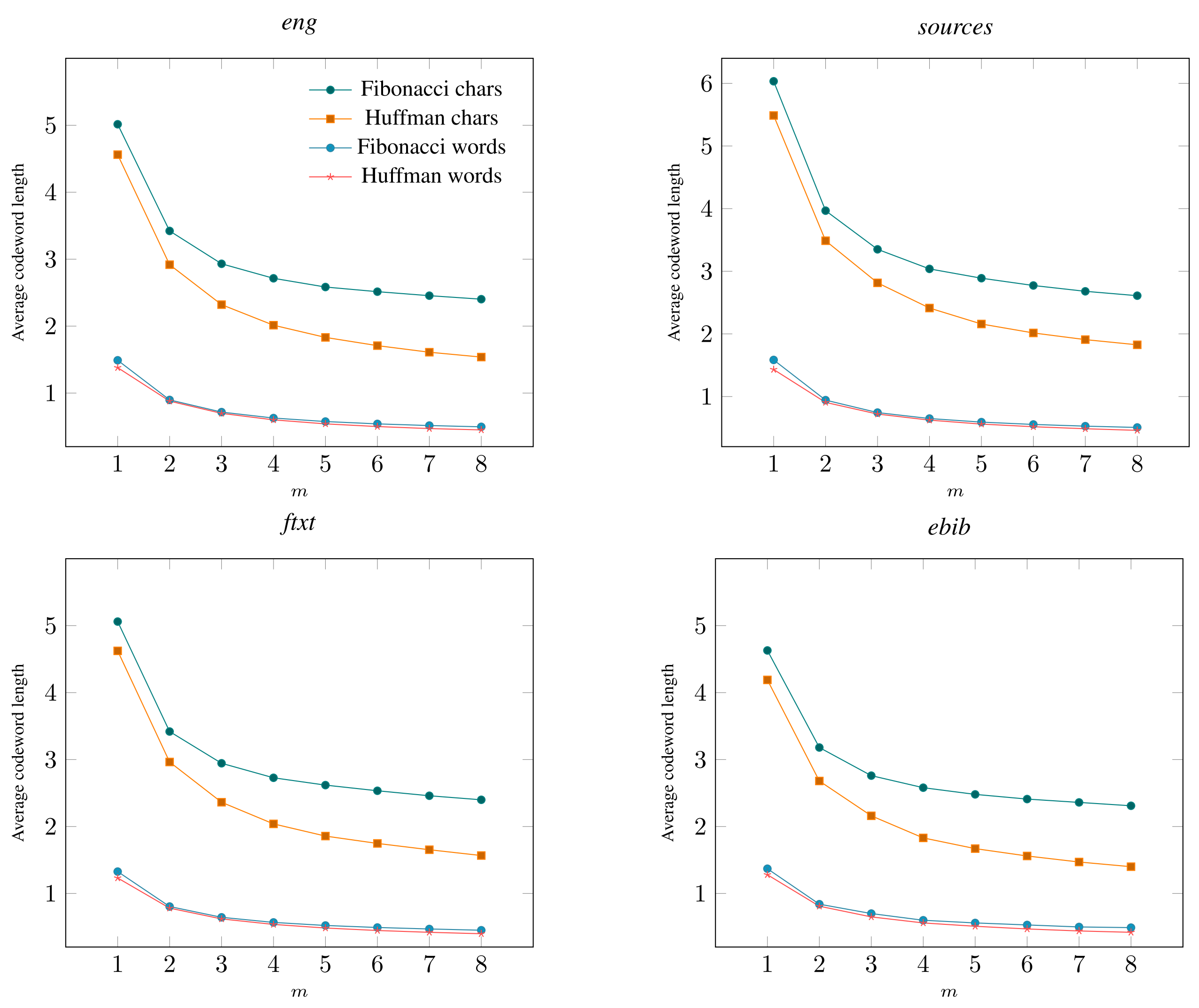


FIGURE 2. Average codeword length as function of *m*.

III. EXPERIMENTAL COMPRESSION RESULTS

To empirically measure the differences in compression gains on real life input data, we considered files in different languages and of different nature, encoded as a sequence of characters as well as a sequence of *words*, so that our alphabets — and the corresponding Huffman trees — are sometimes quite large.

The file *eng* is the concatenation of English text files, selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text; *sources* is formed by C/Java source codes obtained by concatenating all the .c, .h and .java files of the linux-2.6.11.6 distributions; *ftxt* is 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 [32]; and *ebib* is the King James version of the English Bible, in which the text has been stripped of all punctuation signs except blank.

The two first files were downloaded from the Pizza & Chili Corpus.[2] Table 4 presents some statistical details. The second column is the original file sizes in MB, and the other columns give the size of the alphabets in characters and in (different) words.

Table 5 presents comparative compression results on one of the test files, *eng*. The values for the character encodings are bits, trits, or generally $(m+1)$-ary digits per element, and the values for the word based encodings are normalized to show the number of digits per character.

Figure 2 shows the same results in graphical display, but for all our test files mentioned in Table 4. As can be seen, the general behaviour is essentially the same for the different files. Obviously, Huffman coding always achieves better results, but for the alphabet of words, the (red and blue) plots in the lower part of the figures are almost overlapping, implying that the loss of using Fibonacci instead of Huffman can be very small. Taking into account that Fibonacci codes are preferable to Huffman for the generation of the codes, for the decoding and when considering the robustness as an important feature, the new codes presented herein may be a valuable alternative.

IV. FUTURE RESEARCH

In future research, we intend applying these non-binary Fibonacci codes to enhance the compression efficiency of context sensitive flash memory of higher order, extending previous research, see [24] and [2], dealing with the binary case. For instance, in the binary Fibonacci code, every 1-bit is followed by a zero. This has been exploited in [24] to devise a wom code with two writing rounds, in which a Fibonacci encoding is used in a first round, thereby enabling a second round of encoding, overwriting the same bits. In this second round, data is written only on the 0-bits following 1-bits, so that their positions can be detected.

CONFLICT OF INTEREST

None of the authors have a conflict of interest to disclose.

REFERENCES

[1] A. Amir and C. Benson, "Efficient two-dimensional compressed matching," in *Proc. IEEE Data Compress. Conf.*, Jan. 1992, pp. 279–288.

[2] G. Baruch, S. T. Klein, and D. Shapira, "Enhanced context sensitive flash codes," *Comput. J.*, vol. 65, no. 5, pp. 1200–1210, May 2022.

[3] R. N. Brisaboa, A. Farina, G. Navarro, and F. María Esteller, "(S, C)-dense coding: An optimized compression code for natural language text databases," in *Proc. 10th Symp. String Process. Inf. Retr.*, Manaus, Brazil: Springer-Verlag, Oct. 2003, pp. 122–136.

[4] R. N. Brisaboa, E. L. Iglesias, G. Navarro, and R. A. José Param, "An efficient compression code for text databases," in *Advances in Information Retrieval* (Lecture Notes in Computer Science), vol. 2633. Pisa, Italy: Springer-Verlag, 2003, pp. 468–481.

[5] N. R. Brisaboa, S. Ladra, and G. Navarro, "DACs: Bringing direct access to variable-length codes," *Inf. Process. Manage.*, vol. 49, no. 1, pp. 392–404, 2013.

[2]http://pizzachili.dcc.uchile.cl/

[6] Q. Cao, W. Lü, X. R. Wang, X. Guan, L. Wang, S. Yan, T. Wu, and X. Wang, "Nonvolatile multistates memories for high-density data storage," *ACS Appl. Mater. Interfaces*, vol. 12, no. 38, pp. 42449–42471, Sep. 2020.

[7] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash Memories*. Boston, MA, USA: Springer, 1999.

[8] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. IT-21, no. 2, pp. 194–203, Mar. 1975.

[9] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Trans. Algorithms*, vol. 3, no. 2, p. 20, May 2007.

[10] A. Fiat and A. Shamir, "Generalized 'write-once' memories," *IEEE Trans. Inf. Theory*, vol. IT-30, no. 3, pp. 470–479, May 1984.

[11] A. S. Fraenkel, "Systems of numeration," in *Proc. IEEE 6th Symp. Comput. Arithmetic (ARITH)*, Aarhus, Denmark, Jun. 1983, pp. 37–42.

[12] A. S. Fraenkel and S. T. Kleinb, "Robust universal complete codes for transmission and compression," *Discrete Appl. Math.*, vol. 64, no. 1, pp. 31–55, Jan. 1996.

[13] E. N. Gilbert and E. F. Moore, "Variable-length binary encodings," *Bell Syst. Tech. J.*, vol. 38, no. 4, pp. 933–967, Jul. 1959.

[14] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 1–10.

[15] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950.

[16] K. A. S. Immink, *Codes for Mass Data Storage Systems*. Eindhoven, The Netherlands: Shannon Foundation Publishers, 2004.

[17] S. T. Klein and M. K. Ben-Nissan, "On the usefulness of Fibonacci compression codes," *Comput. J.*, vol. 53, no. 6, pp. 701–716, Jul. 2010.

[18] T. Shmuel Klein, C. Tamar Serebro, and D. Shapira, "Non-binary robust universal variable length codes," in *Proc. Data Compress. Conf. (DCC)*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, Eds. Snowbird, UT, USA, Mar. 2020, p. 376.

[19] S. T. Klein and D. Shapira, "Pattern matching in Huffman encoded texts," *Inf. Process. Manage.*, vol. 41, no. 4, pp. 829–841, Jul. 2005.

[20] T. S. Klein and D. Shapira, "Compressed pattern matching in JPEG images," *Int. J. Found. Comput. Sci.*, vol. 17, no. 6, pp. 1297–1306, 2006.

[21] S. T. Klein and D. Shapira, "Huffman coding with non-sorted frequencies," *Math. Comput. Sci.*, vol. 5, no. 2, pp. 171–178, Jun. 2011.

[22] S. T. Klein and D. Shapira, "Compressed matching for feature vectors," *Theor. Comput. Sci.*, vol. 638, pp. 52–62, Jul. 2016.

[23] S. T. Klein and D. Shapira, "Random access to Fibonacci encoded files," *Discrete Appl. Math.*, vol. 212, pp. 115–128, Oct. 2016.

[24] S. T. Klein and D. Shapira, "Context sensitive rewriting codes for flash memory," *Comput. J.*, vol. 62, no. 1, pp. 20–29, Jan. 2019.

[25] G. Longo and G. Galasso, "An application of informational divergence to Huffman codes," *IEEE Trans. Inf. Theory*, vol. IT-28, no. 1, pp. 36–42, Jan. 1982.

[26] W. S. McLaughlin, "Improved distance M-ary (d, k) codes for high density recording," *IEEE Trans. Magn.*, vol. 31, no. 2, pp. 1155–1160, Mar. 1995.

[27] S. W. McLaughlin, J. Luo, and Q. Xie, "On the capacity of M-ary runlength-limited codes," *IEEE Trans. Inf. Theory*, vol. 41, no. 5, pp. 1508–1511, Sep. 1995.

[28] A. Moffat, "Word-based text compression," *Softw., Pract. Exper.*, vol. 19, no. 2, pp. 185–198, Feb. 1989.

[29] A. Moffat and A. Turpin, *Compression and Coding Algorithms* (The International Series in Engineering and Computer Science), vol. 669. Norwell, MA, USA: Kluwer, 2002.

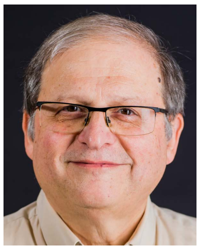
[30] G. Navarro, "Wavelet trees for all," *J. Discrete Algorithms*, vol. 25, pp. 2–20, Mar. 2014.

[31] G. Trogemann, A. Y. Nitussov, and W. Ernst, *Computing in Russia: The History of Computer Devices and Information Technology Revealed*. Wiesbaden, Germany: Vieweg Braunschweig, 2001.

[32] J. Véronis and P. Langlais, "Evaluation of parallel text alignment systems: The arcade project," in *Parallel Text Processing*, J. Véronis, Ed. Norwell, MA, USA: Kluwer, 2000, pp. 369–388, ch. 19.

[33] E. Yaakobi and A. Shpilka, "High sum-rate three-write and nonbinary WOM codes," *IEEE Trans. Inf. Theory*, vol. 60, no. 11, pp. 7006–7015, Nov. 2014.

[34] K. G. Zipf, *Human Behavior and the Principle of Least Effort*. Cambridge, MA, USA: Addison-Wesley, 1949.



SHMUEL T. KLEIN received the B.Sc. degree in mathematics and statistics from the Hebrew University of Jerusalem, in 1978, and the M.Sc. degree in applied mathematics and the Ph.D. degree in computer science from the Weizmann Institute of Science, Rehovot, Israel, in 1981 and 1988, respectively.

Then, he spent three years as a Visiting Assistant Professor at the University of Chicago, and returned, in 1990, to Israel, where he is with the Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel. He has worked on two of the largest full-text information retrieval systems: the Responsa Project at Bar-Ilan University and the Trésor de la Langue Française, University of Chicago. He is currently a Full Professor at Bar-Ilan University, where he is the Chair of the Computer Science Department. He has authored two books, more than 100 refereed journals and conference papers, and 11 patents. His research interests include text processing and in particular data compression.

TAMAR C. SEREBRO received the B.Sc. degree *(cum laude)* in mathematics and computer science from Bar-Ilan University, in 2001, the M.Sc. degree in computer science from Bar-Ilan University in "modeling delta encoding of compressed files," in 2007, and the Ph.D. degree in computer science on "enhancing skeleton Huffman trees," from Ariel University, Israel, in 2021. She holds a teaching position at the Orot College, Israel.

DANA SHAPIRA received the B.Sc. *(cum laude)*, M.Sc. *(magna cum laude)*, and Ph.D. degrees in mathematics and computer science from Bar-Ilan University, Israel, in 1993, 1996, and 2001, respectively. Then, she was a Postdoctoral Fellow with Brandeis University, Waltham, MA, USA. She is currently an Associate Professor with Ariel University, Israel, where she is with the Department of Computer Science. She has coauthored over 40 refereed articles in scientific journals and over 40 refereed articles in conference proceedings and published four patents. She has worked as the Head of the Department of Computer Sciences. She is currently the Head of the M.Sc. degree Program. She has been a Program Committee Member of the Data Compression Conference (DCC), since 2011. Her research interests include data compression, algorithm development, and analysis and scheduling theory.