A Family of Data Compression Codes with Multiple Delimiters

Igor O. Zavadskyi and Anatoly V. Anisimov

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorza@gmail.com

Abstract. A new family of perspective variable length self-synchronazable binary codes with multiple pattern delimiters is introduced. Each delimiter consists of a run of consecutive ones surrounded by zero brackets. These codes are complete and universal. A simple bijective correspondence between natural numbers and any multi-delimiter code set is established. A fast byte aligned decoding algorithm is constructed. Comparisons of text compression rate and decoding speed for different multi-delimiter codes, the Fibonacci code Fib3 and (s, c)-dense codes are also presented.

Keywords: prefix code, Fibonacci code, data compression, robustness, completeness, universality, density, multi-delimiter

1 Introduction

For the last few decades, the data compression technology has accumulated a substantial arsenal of powerful string processing methods. For details, we refer to the book by D. Salomon [11]. The present period of the information infrastructure development actualizes the demand for efficient data compression methods that on one hand provide satisfactory compression rate, and, on the other, support fast encoding, decoding and search in compressed data. Along with this the need for a code robustness in the sense of limiting possible error propagations has also been strengthened.

As is known, the classical Huffman codes provide good compression efficiency approaching to the theoretically best [8]. Unfortunately, Huffman's encoding does not allow the direct search in compressed data by a given compressed pattern. At the expense of losing some compression efficiency this was amended by introducing byte aligned tagged Huffman codes: Tagged Huffman Codes [5], End-Tagged Dense Codes (ETDC) [4] and (s,c)-dense codes (SCDC) [3]. In these methods codewords are represented as sequences of bytes, which along with encoded information incorporate flags for the end of a codeword.

The alternative approach for coding stems from using the Fibonacci numbers of higher orders. The mathematical study of Fibonacci codes was started in the pioneering paper [2]. The authors first introduced families of Fibonacci codes of higher orders with the emphasis on their robustness. Also, they proved completeness of these codes and their universality in the sense of [6].

The most strong argumentation for the use of Fibonacci codes of higher orders in data compression was given in [10]. For these codes, the authors developed fast byte aligned algorithms for decoding and search in the compressed text [9]. They also showed that Fibonacci codes have better compression efficiency comparing with ETDC and SCDC codes for middle size text corpora while still being inferior on decompression and search speed.

Igor O. Zavadskyi, Anatoly V. Anisimov: A Family of Data Compression Codes with Multiple Delimiters, pp. 71–84.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

Another advantage of Fibonacci codes over ETDC, SCDC and Huffman codes is their robustness in the sense of limiting possible error propagations. Although SCDC codes may limit the propagation of errors coursed by bit erroneous inversions, they are completely not resistant to insertions or deletions of bits. Huffman's codes are vulnerable to any of these errors. Whereas in Fibonacci codes errors coursed by a single bit inversion, deletion or insertion cannot propagate over more than two adjacent codewords. In other words, they are synchronizable with synchronization delay at most one codeword.

In this presentation, we study a new family of binary codes with multiple suffix delimiters. These codes were first introduced in [1]. Each delimiter consists of a run of consecutive ones surrounded with zero brackets. Thus, delimiters have the form $01\cdots10$. A number of ones in delimiters is defined by a given fixed set of positive integers $m_1,\ldots,m_t$. The multi-delimiter code $D_{m_1,\ldots,m_t}$ consists of t words $11\cdots10$ with $m_1,\ldots,m_t$ ones and all other words in which delimiters occur only as a suffix. For example, the multi-delimiter code $D_{2,3}$ consists of words 110, 1110 and all other words in which 110 or 1110 occurs only as a suffix, e.g. 0110, 01110, 10110, etc.

By their properties, the multi-delimiter codes are close to the Fibonacci codes of higher orders. Due to robust delimiters, multi-delimiter codes are synchronizable with synchronization delay at most one codeword, as well as Fibonacci codes. We prove completeness and universality of such codes. There also exists a bijection between any code $D_{m_1,\ldots,m_t}$ and the set of natural numbers. This bijection is implemented by very simple encoding and decoding procedures. For practical use we present a byte aligned decoding algorithm with better computational complexity than that of Fibonacci codes.

Each of ETDC, SCDC, Fibonacci and multi-delimiter codes is well suited for natural language text compression if words of a text are considered as atomic symbols. As shown in [10], the Fibonacci code of order three, denoted by Fib3, has the best compression rate when applied to this kind of data. From our study, it follows that the simple code D_2 with one delimiter 0110 has asymptotically higher density as against Fib3, although it is slightly inferior in compression rate for realistic alphabet sizes of natural language texts.

We also note that by varying delimiters for better compression we can adapt multi-delimiter codes to a given probability distribution and an alphabet size. Thus, for example, we compare the codes $D_{2,3}$, $D_{2,3,5}$ and $D_{2,4,5}$ with the code Fib3. Those multi-delimiter codes are asymptotically less dense than Fib3. Nevertheless, in practice the alphabet size of a text is often relatively small, from a few thousand up to a few million words. For such texts the aforementioned multi-delimiter codes outperform the Fib3 code in compression rate by $2-3\%$, while both Fibonacci and multi-delimiter codes significantly outperform the ETDC/SCDC codes.

The structure of the presentation is as follows. In Section 2 we define the family of multi-delimiter codes and discuss their density. A bijective correspondence between the set of natural numbers and codewords of any code $D_{m_1,\ldots,m_t}$ is established in the next section. Also, herein we present the bitwise encoding/decoding algorithms. The completeness and universality of multi-delimiter codes is proven in Section 4. The fast byte aligned decoding algorithm for the multi-delimiter code $D_{2,3,5}$ is given in Section 5. This code appears to be the most efficient in compression among all multi-delimiter codes when applied to small or mid-size texts. In Section 6 we present the results of computational experiments to compare the compression rate and decoding time of

Index	Fib2	D_1	$D_{1,2}$	Fib3	D_2	$D_{2,3}$	$D_{2,3,4}$
1	11	10	10	111	110	110	110
2	011	010	010	0111	0110	0110	0110
3	0011	0010	110	00111	00110	1110	1110
4	1011	00010	0010	10111	10110	00110	00110
5	00011	11010	0110	000111	000110	10110	10110
6	01011	000010	00010	010111	010110	01110	01110
7	10011	011010	00110	100111	100110	000110	11110
8	000011	110010	000010	110111	0000110	010110	000110
9	001011	111010	000110	0000111	0010110	100110	010110
10	010011	0000010	111010	0010111	0100110	001110	100110
11	100011	0011010	0000010	0100111	1000110	101110	001110
12	101011	0110010	0000110	1000111	1010110	0000110	101110
13	0000011	1100010	0111010	1010111	1110110	0010110	011110
14	0001011	0111010	1110010	0110111		0100110	0000110
15	0010011	1110010	1110110	1100111		1000110	0010110
16	0100011	1111010	1111010			1010110	0100110
17	1000011					0001110	1000110
18	0101011					0101110	1010110
19	1001011					1001110	0001110
20	1010011						0101110
21							1001110
22							0011110
23							1011110

Table 1. Sample codeword sets of multi-delimiter and Fibonacci codes

SCDC, Fibonacci and multi-delimiter codes. And in the last section we summarize the advantages of multi-delimiter codes.

2 Definition of multi-delimiter codes

Let $\mathcal{M} = \{m_1, \ldots, m_t\}$ be a set of integers, given in ascending order, $0 < m_1 < \cdots < m_t$.

Definition 1 *The multi-delimiter code $D_{m_1,\ldots,m_t}$ consists of all the words of the form $1^{m_i}0, i = 1, \ldots, t$ and all other words that meet the following requirements:*

(i) for any $m_i \in \mathcal{M}$ a word does not start with a sequence $1^{m_i}0$;
(ii) a word ends with the suffix $01^{m_i}0$ for some $m_i \in \mathcal{M}$;
(iii) for any $m_i \in \mathcal{M}$ a word cannot contain the sequence $01^{m_i}0$ anywhere, except a suffix.

The given definition implies that code delimiters in $D_{m_1,\ldots,m_t}$ are sequences of the form $01^{m_i}0$. However, the code also contains shorter words of the form $1^{m_i}0$, which form a delimiter together with the ending zero of a preceding codeword.

The sample of codewords of the length ≤ 7 for some multi-delimiter codes and, for comparison, some Fibonacci codes is given in Table 1. As is seen, the codes $D_{2,3}$ and $D_{2,3,4}$ with 2 and 3 delimiters respectively contain many more short codewords than both the Fibonacci code Fib3 and the one-delimiter code D_2. This is an important factor when considering the compression efficiency.

We calculate the number of short codewords for several multi-delimiter codes that are potentially suitable for natural language text compression. Also, we calculate

Code	The number of codewords of length $\leq n$								
	Asymptotic	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$	$n=15$
The codes with the shortest codeword of length 2									
Fib2	1.618^n	1	2	4	7	12	20	33	986
D_1	1.755^n	1	2	3	5	9	16	28	1432
$D_{1,2}$	1.618^n	1	3	5	7	10	16	27	799
$D_{1,3}$	1.674^n	1	2	4	7	11	18	30	1106
The codes with the shortest codeword of length 3									
Fib3	1.839^n	0	1	2	4	8	15	28	2031
D_2	1.867^n	0	1	2	4	7	13	24	1906
$D_{2,3}$	1.785^n	0	1	3	6	11	19	33	1874
$D_{2,4}$	1.823^n	0	1	2	5	9	17	30	1998
$D_{2,5}$	1.844^n	0	1	2	4	8	15	28	1999
$D_{2,3,4}$	1.731^n	0	1	3	7	13	23	39	1721
$D_{2,4,5}$	1.796^n	0	1	2	5	10	19	34	2019
$D_{2,4,6}$	1.809^n	0	1	2	5	9	18	32	2032
The codes with the shortest codeword of length 4									
Fib4	1.928^n	0	0	1	2	4	8	16	1606
D_3	1.933^n	0	0	1	2	4	8	15	1510

Table 2. The number of codewords of multi-delimiter and Fibonacci codes

the asymptotic densities of these codes using the standard technique of generating functions. The results are presented in Table 2.

In general, codes with more delimiters contain more short words, although they have worse asymptotic density. This regularity is also related to lengths of delimiters: the shorter they are the larger quantity of short words a code contains. Considering the application for text compression, the most efficient seems to be the codes $D_{2,\ldots}$, which we thoroughly investigate.

3 Encoding integers

We define a multi-delimiter code as a set of words. There exists a simple bijection between this set and the set of natural numbers. This bijection allows us to encode integers.

Let $\mathcal{M} = \{m_1, \ldots, m_t\}$ be the set of parameters of the code $D_{m_1,\ldots,m_t}$. By $N_{\mathcal{M}} = \{j_1, j_2, \ldots\}$ denote the ascending sequence of all natural numbers that do not belong to $\mathcal{M}$.

By $\varphi_{\mathcal{M}}(i)$ denote the function $\varphi_{\mathcal{M}}(i) = j_i$, $j_i \in N_{\mathcal{M}}$ as defined above.

It is easy to see that the function $\varphi_{\mathcal{M}}$ is a bijective mapping of the set of natural numbers onto $N_{\mathcal{M}}$. Evidently, this function and the inverse function $\varphi_{\mathcal{M}}^{-1}$ can be constructively implemented by simple constant time procedures.

A run of consecutive ones in a word w is called *isolated* if it is a prefix of this word ending with zero, or it is its suffix starting with zero, or it is a substring of w surrounded with zeros, or it coincides with w.

The main idea of encoding integers by the code $D_{m_1,\ldots,m_t}$ is as follows. We scan the binary representation of an integer from left to right. During this scan each isolated group of i consecutive 1s is changed to $\varphi_{\mathcal{M}}(i)$ isolated 1s. This way we exclude the appearance of delimiters inside a codeword. In decoding, we change internal isolated groups of j consecutive 1s to groups of $\varphi_{\mathcal{M}}^{-1}(j)$ ones. The detailed description of the encoding procedure is as follows.

Bitwise Integer Encoding Algorithm

Input: an integer $x = x_n x_{n-1} \cdots x_0$, $x_i \in \{0,1\}, x_n = 1$;
Result: the corresponding codeword from $D_{m_1,\ldots,m_t}$.

1. $x \leftarrow x - 2^n$, i.e. extract the most significant bit of the number x, which is always 1.
2. If $x = 0$, append the sequence $1^{m_1}0$ to the string $x_{n-1} \cdots x_0$, which contains only zeros or empty. *Result* $\leftarrow x_{n-1} \cdots x_0 1^{m_1} 0$. Stop.
3. If the binary representation of x takes the form of a string $0^r 1^{m_i} 0, r \geq 0, m_i \in \mathcal{M}, i > 1$, then *Result* $\leftarrow x$. Stop.
4. In the string x replace each isolated group of i consecutive $1s$ with the group of $\varphi_{\mathcal{M}}(i)$ consecutive $1s$ except its occurrence as a suffix of the form $01^{m_i}0, i > 1$. Assign this new value to x.
5. If the word ends with a sequence $01^{m_i}0, i > 1$, then *Result* $\leftarrow x$. Stop.
6. Append the string $01^{m_1}0$ to the right end of the word. Assign this new value to x. *Result* $\leftarrow x$. Stop.

According to this algorithm, if $x \neq 2^n$, the delimiter $01^{m_1}0$ with m_1 ones does not contain information bits, and therefore it should be deleted during the decoding. However, delimiters of the form $01^{m_i}0, i > 1$ are informative parts of codewords, and they must be processed during the decoding. If $x = 2^n$, the last $m_1 + 1$ bits of the form $1^{m_1}0$ must be deleted.

Bitwise Decoding Algorithm

Input: a codeword $y \in D_{m_1,\ldots,m_t}$.
Result: the integer given in the binary form which encoding results in y.

1. If the codeword y is of the form $0^p 1^{m_1} 0$, where $p \geq 0$, extract the last $m_1 + 1$ bits and go to step 4.
2. If the codeword y ends with the sequence $01^{m_1}0$, extract the last $m_1 + 2$ bits. Assign this new value to y.
3. In the string y replace each isolated group of i consecutive $1s$, where $i \notin \mathcal{M}$, with the group of $\varphi_{\mathcal{M}}^{-1}(i)$ consecutive $1s$. Assign this new value to y.
4. Prepend the symbol 1 to the beginning of y. *Result* $\leftarrow y$. Stop.

Let us give the example. We encode the number $14 = 1110_2$ using the code $D_{2,3}$. For this code, $\mathcal{M} = \{2,3\}$, $N_{\mathcal{M}} = \{1,4,5,\ldots\}$, $\varphi_{\mathcal{M}}(2) = 4$, $\varphi_{\mathcal{M}}(3) = 5$, $\varphi_{\mathcal{M}}(4) = 6$ etc.

1. Extracting the most significant bit we obtain the number 110.
2. 110 is the isolated group of ones. Replace it with the isolated group of $\varphi_{\mathcal{M}}(2) = 4$ ones, i.e. 11110.
3. Appending the string $01^{m_1}0$ to the right end of the word we get the result 111100110.

Now let us decode the codeword 111100110.

1. Extracting the last $m_1 + 2$ bits we obtain the number 11110.
2. Replace the isolated group of 4 ones in the beginning of the codeword with the isolated group of $\varphi_{\mathcal{M}}^{-1}(4) = 2$ ones: 110.
3. Prepend the symbol 1 to the beginning of the word: 1110.

4 Some general properties of multi-delimiter codes

Evidently, any multi-delimiter code is prefix-free and thus uniquely decodable (UD). However, this fact can be proved formally by checking the Kraft inequality. If it holds as the equality, the code is also complete, which means that no codeword can be added to a code in a way that preserves the UD property.

Theorem 1. *Each multi-delimiter code $D_{m_1,\ldots,m_t}$ is uniquely decodable, complete and universal.*

Proof. Completeness and UD property of any multi-delimiter code $D_{m_1,\ldots,m_t}$ is proved in [1]. The proof is based on checking the Kraft equality.

Let us prove the universality of multi-delimiter codes. The notion of universality was introduced by P. Elias [6] to reflect the property of a code to be nearly optimal for data sources with any given probability distribution. Formally this means that there exists a constant K such that for any finite distribution of probabilities $P = (p_1, \ldots, p_n)$, where $p_1 \geq p_2 \geq \ldots$, the inequality $\sum_{i=1}^{n} l_i p_i \leq K \cdot \max(1, E(P))$ holds true, where $E(P) = -\sum_{i=1}^{n} p_i \log_2 p_i$ is entropy of distribution P and l_i are codeword lengths.

Note that encoding procedure that transforms a number x into the corresponding codeword of the code $D_{m_1,\ldots,m_t}$ can enlarge each internal isolated group of sequential 1s in the binary representation of x to a maximum of t ones. The quantity of such groups does not exceed $\frac{1}{2}\log_2 x$. To some binary words the delimiter $01^{m_1}0$ could be externally appended, while the leftmost 1 is always deleted. Therefore the length of the codeword is upper bounded by the value $(\frac{1}{2}t + 1)\log_2 x + m_1 + 1$.

Let us sort codewords from $D_{m_1,\ldots,m_t}$ in ascending order of their bit lengths: $a_1, a_2, \ldots$ We map them to symbols of the input alphabet sorted in descending order of their probabilities. Evidently, the correspondence between an integer and the length of its codeword is not monotonic. Nevertheless, there are at least i words of lengths that do not exceed the upper bound for i, which is equal to $(\frac{1}{2}t + 1)\log_2 i + m_1 + 1$. Thus, the length of a_i does not exceed this bound too. To conclude the proof it only remains to apply general Lemma 6 by Apostolico and Fraenkel taken from [2]. "Let ψ be a binary representation such that $|\psi(k)| \leq c_1 + c_2 \log k$ $(k \in \mathbb{Z}^+)$, where c_1 and c_2 are constants and $c_2 > 0$. Let p_k be the probability to meet k. If $p_1 \geq p_2 \geq \cdots \geq p_n$, $\Sigma p_i \leq 1$ then ψ is universal." $\square$

5 Fast decoding

The value of a code depends not only on compression rate, but on a number of other properties. And not least of all it concerns the time of compression and decompression. The decompression time is more critical than the time of compression. That is why in this presentation we only concentrate on the accelerating of decoding.

The aforementioned encoding and decoding algorithms are bitwise, and thus, they are quite slow. To accelerate them we construct a byte aligned lookup table method, which performs the same mapping as the bitwise decoding algorithm. The main idea of the proposed method is similar to that developed in [10]. At each iteration, the algorithm processes some parameters from a table row, which examples are given in Table 3. The choice of a row depends on two parameters listed in the left two columns of the table. They are a byte read from the input (column 2) and a value r

which depends on bits left unprocessed at the previous iteration (column 1). These two parameters can be considered as indices of the two-dimensional array TAB containing all decoded numbers which can be extracted from a current byte and also some other parameters.

As shown, the code $D_{2,3,5}$ has one of the best compression rates comparing with other multi-delimiter codes. Therefore, for this code we give the detailed description of the decoding algorithm. We consider the simplest one byte variant, i.e. processing 8 bits per iteration. It is not difficult to extend considered constructions to any other multi-delimiter code and to other number of bytes. The table-driven decoding algorithm is given below. Its parameters have the following meanings:

w_1, w_2, w_3, w_4 - decoded numbers that can be extracted at the current iteration.
l_1 - the bit length of a number w_1.
g - the number of codewords for which decoding is finished at the current iteration.
w - a partially decoded number. We use this variable to transfer decoded bits from iteration to iteration when some codeword is split among bytes.
r_{prev} - an index, which depends on the bits left unprocessed at the previous iteration.
r - an index, which depends on the bits left unprocessed at the current iteration.
$Text$ - a coded text.
$Dict$ - the dictionary that maps the decoded numbers to the words of the input text.
TAB - the array containing values dependent on the remainder r_{prev} and the next byte of the code.

Fast Byte Aligned Decoding Algorithm

Input: a coded $Text$.
Result: the sequence of integers.

```
w ← 1, r_prev ← 0, i ← 0
while i < length of encoded text
    (g, w1, w2, w3, w4, r, l1) ← TAB[r_prev][Text[i]]
    w ← (w << l1)|w1        // append the w1 bits to the right of w
    if g > 0
        output Dict(w)
        if g > 1
            output Dict(w2)
            if g > 2
                output Dict(w3)
                w ← w4
            else
                w ← w3
        else
            w ← w2
    r_prev ← r
    i ← i + 1
```

Let us explain how this algorithm works. A byte being processed is divided into two parts: the left one contains bits, which can be decoded unambiguously, and the right part contains the rest of the byte. The result of decoding of the left part is

r_{prev}	$Text[i]$	g	w_1	w_2	w_3	w_4	l_1	r
0	11000 111	1		100			0	6
6	01101 011	2	1110	1	1		4	2
2	11100110	2	0111110	10	1		7	0
0	10100 011	0	10100				5	5
5	01100110	3		1	10	1	0	0

Table 3. Rows of the lookup table

assigned to variables w_1, w_2, w_3 and w_4 (since the length of the shortest codeword of $D_{2,3,5}$ is 3 bits, one byte cannot contain more than 4 adjacent codewords or their parts). If some byte contains parts of i codewords, the first part might contain only the ending of some codeword, while the last one might contain only the beginning of a codeword. This beginning is stored in the column w_i of the table TAB and it is assigned to the variable w. At the beginning of the next iteration, we append a new value w_1 to the right of the bit representation of w (line 4). This is quite a simple operation if we know the bit length of w_1, which is stored in the column l_1.

If there are no bits that can be decoded unambiguously in the last number w_i in the byte, we assign 1 to w_i, since the decoded number should always be prepended by the leftmost '1' bit (see the last step of the bitwise decoding algorithm). If the ending of the last codeword w_i coincides with the ending of the byte (it implies that $i = g$), we create the fictitious codeword w_{i+1} which is equal to 1. Such situation is illustrated by rows 3 and 5 of Table 3. For the same reason we assign 1 to w at the beginning of the algorithm. The whole Table 3 shows the rows of TAB array used for decoding the text 11000111 01101011 11100110 10100011 01100110. The unambiguously decoded bits are separated from the rest bits with spaces.

Of course, the decoding should be performed with regard to the right part of the previous byte, which contains bits that cannot be decoded unambiguously. That is, if some byte begins with bits 10, it is decoded differently when the previous byte ends with 01 and when it ends with 011. Indeed, in the first case the codeword delimiter 0110 appears, while in the second case we have the sequence 01110, which cannot appear at the end of a codeword. However, it follows from the bitwise decoding algorithm that each zero bit clears the decoding history. More precisely, if we process the code $D_{2,3,5}$ bit-by-bit from left to right and match the sequence 10 or 00, in both cases we can decode the first of these two bits unambiguously regardless the bits right to them. Therefore, all bits of some byte, starting from the left and up to the bit preceding the rightmost zero, can be decoded unambiguously. Regarding the rightmost zero bit, it can be decoded unambiguously in the following cases.

1. Some codeword ends with this zero.
2. This zero belongs to the sequence $0 \cdots 0$, which is the prefix part of some codeword.
3. The byte contains 3 or more ones after this zero.

In all other cases, the rightmost zero either might belong or not belong to the delimiter 0110. If it belongs to this delimiter, it should be discarded together with the whole delimiter. Otherwise, it should be present in the decoded number. These two cases can be distinguished only at the next iteration.

Also, we note that if a byte ends with the run of 6 ones, the first two bits of this byte do not affect the next ensuing decoding since any of these ones cannot belong to a delimiter.

Value of r (type)	Number of ones in the end of a byte	Is the rightmost zero bit decoded?
0	0	yes
1	0	no
2	1	yes
3	1	no
4	2	yes
5	2	no
6	3	yes
7	4	yes
8	5	yes
9	≥ 6	yes

Table 4. The types of the byte endings in the fast decoding

Thus, we have 10 types of byte endings, which differently affect the next byte decoding. These types are listed in Table 4 and correspond to 10 possible values of r.

Now we can calculate the space complexity of the byte aligned decoding algorithm. It is easy to show that the value w_1 cannot be longer than 11 bits and each of the other values w_i fit into one byte. Thus, if for each value we use a whole number of bytes, then one row of the array TAB could be stored in 8 bytes and the whole array requires $8 \times 10 \times 256$ bytes =20K memory. However, on the bit level each row of the array TAB can be packed only into 4 bytes. For such representation, we have built more sophisticated, but several times faster implementation of the table decoding algorithm in assembly language (its details are out of the scope of this presentation). In such case 10K memory needed to store the array TAB. For comparison, the fastest one-byte table decoding algorithm for Fib3 code reported in [10] requires 21,4K memory for precomputed arrays.

If the code is applied to represent a sequence of numbers, one need only store the array TAB. However, if it is used for compressing many other data types, the dictionary also should be stored. The application of multi-delimiter codes to natural language text compression is discussed in the next section. Also, the experimental estimates of the time complexity of the fast decoding algorithm are presented.

6 Data compression by multi-delimiter codes

To determine the data compression efficiency of a code, first of all it is useful to calculate the number of codewords of the length not greater than n. The corresponding results are presented in Table 2. As is seen, one-delimiter codes D_{m-1} are asymptotically denser than Fibonacci codes Fib$m$ although they contain less short codewords. As we add other delimiters to a code, the asymptotic density decreases, while the number of short codewords increases. In general, the multi-delimiter codes family is more adaptive as against Fibonacci codes. Choosing appropriate values of $m_1, \ldots, m_t$ allows us to tightly approach the code $D_{m_1,\ldots,m_t}$ to the specific distribution of input symbols and their alphabet size.

For natural language text compression, as noted above, the most efficient seem to be codes with the shortest delimiter 0110. The "champions" are the codes $D_{2,3}$, $D_{2,3,4}$, $D_{2,3,5}$ and $D_{2,4,5}$. However, the code $D_{2,3,4}$ has quite low asymptotic density, which narrows its application to only small alphabets. We investigate more thoroughly the other three codes.

Before presenting the experimental results, let us discuss one specific property of multi-delimiter codes, which relates to use a dictionary in the decoding. In particular, this relates to decoding of natural language texts.

All the encoding/decoding algorithms we discussed fit the following schema. During the encoding, the mapping (*word of text*, *codeword*) is used, where the words of a text are sorted in descending order of frequency, while the codewords are sorted in ascending order of codeword lengths. The decoding process is reverse: one should construct a mapping from the set of codewords to the set of text words. In order to fasten the decoding, a data structure with low access time should be used to store the words of a text. For these purposes, the most efficient data structure is the array with integer indices. It allows us to access the words in a $\mathit{Dictionary}[i]$ style, that is $*(\mathit{Dictionary} + i)$ in C notation. This only requires one addition and three memory readings to obtain a word; however, it also requires constructing a mapping between the set of codewords and the set of integer indices. For Fibonacci codes such mapping can be efficiently performed using some remarkable properties of Fibonacci numbers (the method is developed in [10]); in the SCDC decoding the arithmetic properties of the codes are utilized.

For multi-delimiter codes, we described the encoding and decoding mappings in Section 3. Denote them by ψ and ψ^{-1} respectively. However, they have one essential disadvantage: the codewords $\psi(1), \psi(2), \ldots$ are not sorted in ascending order of their lengths. It follows that the words of a text in the array *Dictionary* could be ordered not in descending order of frequencies $f(w_i)$. This is not a problem since the main ordering principle holds: if $f(w_i) > f(w_j)$, then the length of the codeword $\psi(w_i)$ is equal or less than the length of the codeword $\psi(w_j)$. However, the problem is that the codewords $\psi(1), \ldots, \psi(n)$ do not constitute the set of n shortest codewords. We see three ways to resolve this issue, which represents the trade-off between time, space, and compression efficiency.

1. Encode the text using the codewords $\psi(1), \ldots, \psi(n)$, i.e. not the shortest codewords. As the computational experiment shows, this decreases the compression rate up to 2% but does not increase the time and space complexity of the decoding.
2. Enlarge the size of the dictionary to some value $k > n$ and assign the values to its elements with the indices $\psi^{-1}(c_1), \ldots, \psi^{-1}(c_n)$, where $c_1, \ldots, c_n$ is the set of n shortest codewords, so that $\psi^{-1}(c_i) < k, 1 \leq i \leq n$. The enlarged dictionary is sparse since $k-n$ elements are empty. This requires more memory for the decoding but does not enlarge the size of the dictionary that should be transmitted to a recipient along with the encoded file, because only the set of text words ordered according to their frequencies has to be transmitted. For the code $D_{2,3,5}$, it is enough to increase the *Dictionary* array to four times its original size to achieve the compression less than 0.1% away from the optimal for this code. However, the actual memory consumption increases less than three times, since the enlarged array is sparse. In this case, the decoding time remains optimal.
3. Build the array of some fixed length t for the words with higher frequencies and store the other words in a map (*number*, *word of text*). The non-empty elements of the array have the indices that correspond to shortest codewords. In this case, the access to the map is rather longer, but this data structure is not sparse. If we choose a value of t so that the space complexity is increased by 10%, the time is increased approximately twice. However, the compression rate remains optimal.

Text	Words	Dictionary size	Entropy bits	SCDC	Fib3	$D_{2,3,5}$	$D^L_{2,3,5}$	$D_{2,3}$	$D_{2,4,5}$
Hamlet, Shakespeare	30 694	4 501	9.2112	10.47 13.7%	10.01 8.7%	9.76 6.0%	10.05 9.1%	9.83 6.7%	9.85 6.9%
Text in Ukrainian	90 691	14 879	10.6455	12.04 13.1%	11.4 7.1%	11.26 5.8%	11.61 9.1%	11.35 6.6%	11.33 6.4%
Robinson Crusoe, D. Defoe	121 325	5 994	8.73519	10.13 16.0%	9.41 7.7%	9.13 4.5%	9.31 6.6%	9.13 4.5%	9.21 5.4%
Bible, KJV	779 079	12 452	8.6279	10.138 17.5%	9.219 6.9%	8.954 3.8%	9.05 4.9%	9.044 4.8%	9.071 5.1%
Articles from Wikipedia	19 507 783	288179	11.0783	12.869 16.2%	11.564 4.4%	11.492 3.7%	11.544 4.2%	11.488 3.7%	11.471 3.5%

Table 5. Average codeword lengths and excesses over entropy bits for Fib3 and some multi-delimiter codes

In our computational experiments, we follow the second approach by default since usually in natural language texts the size of a vocabulary is never greater than a few megabytes. For modern computers related RAM overheads are quite acceptable. However, some results of the first approach are also presented for comparison.

The results of experiments on compression efficiency of different codes are shown in Table 5. Compression efficiency of SCDC, the Fibonacci code Fib3 and multi-delimiter codes is measured for the texts of different size in two languages: 4 texts in English and 1 in Ukrainian. The largest corpus contains the articles randomly chosen from the English Wikipedia. The punctuation signs in the texts are ignored; lowercase and uppercase symbols are not distinguished. For each text the values of s and c giving the best compression rate of SCDC are determined. Also the "RAM economy" version of the code $D_{2,3,5}$ is tested, which does not enlarge dictionary array (the results are in the column $D^L_{2,3,5}$). The word-level entropy of the texts is calculated. The compression rate is presented as the average codeword length in bits (the first value in a cell) and also as the excess over the entropy bound in percents.

As seen, the multi-delimiter and Fibonacci codes significantly outperform the SCDC codes by compression rate. And codes with 3 delimiters, in its turn, perform $1.2 - 1.8$ times closer to the entropy bound than the Fib3. Among all tested multi-delimiter codes, $D_{2,3,5}$ demonstrates the best compression rate for small and mid-size texts (up to 1M words), but for the large text of 19M words the code $D_{2,4,5}$ becomes slightly better. That is to be expected, since $D_{2,4,5}$ has the better asymptotic density as shown in Table 2. The code $D_{2,3}$ is superior to Fib3, and this shows the usefulness of a second delimiter, but it is inferior to $D_{2,3,5}$ or $D_{2,4,5}$, which demonstrates the benefit of a third delimiter. The rest of multi-delimiter codes presented in Table 2 are inferior at least to one of these three tested codes for all five texts. However, the code $D_{2,3,4}$ shows the best performance on some extremely small texts, which alphabets are less than 2000 words, but, in general, this is too small size for natural language text compression.

The excesses over entropy bounds for the codes Fib3, $D_{2,3}$, $D_{2,4,5}$ and $D_{2,3,5}$ are also shown in Fig. 1.

We also compared the decompression time for three codes: the optimal (s, c)-codes, Fib3 and $D_{2,3,5}$. We applied the one-byte variant of the fastest byte-aligned decoding method described in [10] (with two precomputed tables and no multiplications) for Fib3, the bitwise and byte-aligned decoding algorithms for $D_{2,3,5}$ described

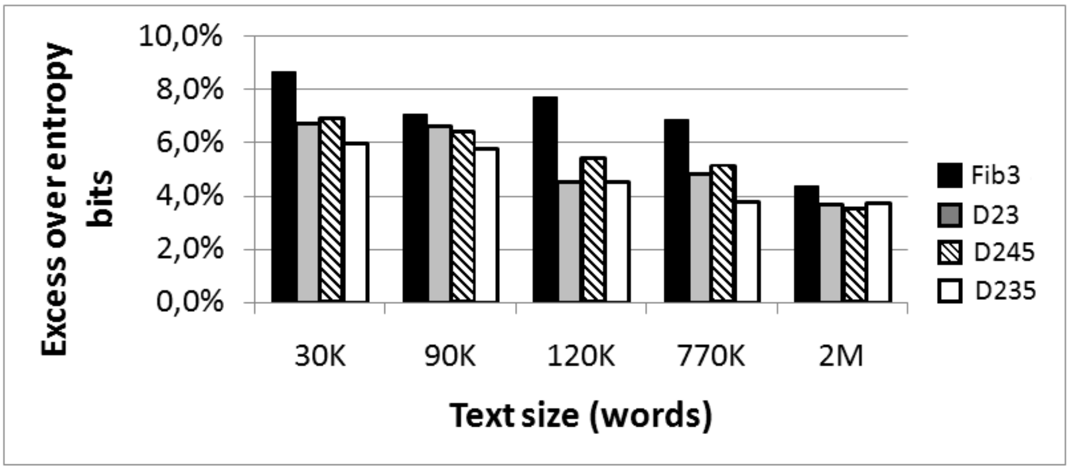

Figure 1. Excesses over entropy bits for the codes Fib3, $D_{2,3}$, $D_{2,4,5}$, $D_{2,3,5}$.

Text	SCDC algorithm	$D_{2,3,5}$, byte-aligned algorithm	$D_{2,3,5}$, bitwise algorithm	Fib3, byte-aligned algorithm
Robinson Crusoe, D. Defoe	15.1	17.3 14.6%	48.2 219.2%	31.2 106.6%
Bible, KJV	103	111 7.8%	270 162.1%	200 94.2%

Table 6. Empirical comparison of decoding time, in milliseconds

in sections 3 and 5, respectively. The values are averaged over 1000 runs of decoding on a PC with AMD Athlon II X2 245 2.9 GHz processor, 4 GB RAM, running Windows 7 32-bit operating system. The result of decompression is stored in RAM as the array of words; the time needed to write this array to file is excluded since this is too expensive operation and it dissolves the differences between decompression methods themselves. The results for two texts in English are presented in Table 6. Values are given in milliseconds and the overrun comparing to SCDC in percents is also presented.

As seen, for the code $D_{2,3,5}$ the fast byte-aligned decoding performs significantly faster than that of the Fib3 code. This is expected, since the fast decoding algorithm for $D_{2,3,5}$ performs on average many fewer operations to obtain the index of a word in the dictionary (lines $4-15$), while the reading from the precomputed array (line 3) is roughly of the same time as the similar operation in the fast Fib3 decoding. However, the byte-aligned decoding algorithm for $D_{2,3,5}$ remains slightly inferior to SCDC decoding.

The code Fib3, in comparison with the multi-delimiter codes, also has a drawback, which refers to the characteristic of the instantaneous separation that is important for searching a word in a compressed file without its decompression. As Fib3, so multi-delimiter codes as well as many other codes used for text compression are characterized by the following: for any codeword w, if a bit sequence w occurs in a compressed file, we can not guarantee that it truly corresponds to the occurrence of the whole codeword w. It could be a suffix of another codeword or it could contain another word as a suffix. In multi-delimiter codes, to check if w is truly a separate codeword, it is enough to consider the fixed number of bits that precede w. For example, it is enough to check four bits for the code D_2. If they turn out to be 0110, then w is a codeword, otherwise it is not. However, it is not enough to check any fixed number of bits preceding a codeword in the code Fib3, since the delimiter and the shortest word in this code is 111. Several such codewords can "stick together" if they are adjacent.

This property of multi-delimiter codes allows to perform the pattern search in a compressed file a bit faster. However, we do not discuss the search problem in this presentation in details. General binary search methods (e.g., [9], [7]) can be applied to multi-delimiter codes as well.

7 Conclusion

We introduce a new family of variable length prefix multi-delimiter codes. They possess all properties known for the Fibonacci codes such as completeness, universality, simple vocabulary representation, and strong robustness. But also they have some more advantages:

(i) Adaptability. Varying delimiters we can adapt a multi-delimiter code to a given source probability distribution and an alphabet size.
(ii) The better compression rate for natural language text compressing.
(iii) The faster byte aligned decoding method.
(iv) Instantaneous separation of codewords allowing faster compressed search.

The multi-delimiter codes seem to be preferable over (s, c) dense codes in the compression of small and mid-size natural language texts, since they have significantly better compression rate but only slightly greater decompression time. These codes together with the Fibonacci codes can be useful in many practical applications.

References

1. A. Anisimov and I. Zavadskyi: *Variable length prefix (δ, k)-codes*, in Proc. IEEE International Black Sea Conference on Communications and Networking, BlackSeaCom'15, Constanta, Romania, 2015, pp. 43–47.
2. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using fibonacci representations*. IEEE Transactions Information Theory, 33 1987, pp. 238–245.
3. N. Brisaboa, A. Farina, G. Navarro, and M. Esteller: *(s,c)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, no. 2857 in Lecture Notes in Computer Science, Manaus, Brazil, 2003, Springer-Verlag, Berlin, pp. 122–136.
4. N. Brisaboa, E. Iglesias, G. Navarro, and J. Parama: *An efficient compression code for text databases*, in 25th European Conference on IR Research, no. 2633 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 468–481.
5. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text*. ACM Transactions on Information Systems, 18(2) 2000, pp. 113–119.
6. P. Elias: *Universal codeword sets and representations of the integers*. IEEE Transactions Information Theory, 21 1975, pp. 194–203.
7. S. Faro and T. Lecroq: *An efficient matching algorithm for encoded dna sequences and binary strings*, in Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching, CPM'09, no. 5577 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 106–115.
8. D. Huffman: *A method for the construction of minimum-redundancy codes*. Proc. IRE, 40 1952, pp. 1098–1101.

9. S. T. Klein and M. Ben-Nissan: *Accelerating boyer moore searches on binary texts*, in Proc. Intern. Conf. on Implementation and Application of Automata, CIAA'07, no. 4783 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 130–143.
10. S. T. Klein and M. Ben-Nissan: *On the usefulness of fibonacci compression codes.* Computer Journal, 53(6) 2010, pp. 701–716.
11. D. Salomon: *Variable-Length Codes for Data Compression*, Springer-Verlag, London, U.K., 2007.