Information sciences. 2012, vol. 183, issue 1, p. 66-91. http://dx.doi.org/10.1016/j.ins.2011.06.019

Fast Decoding Algorithms for Variable-Lengths Codes

Jiří Walder[a], Michal Krátký[a], Radim Bača[a], Jan Platoš[a], Václav Snášel[a]

[a] *Department of Computer Science*
VŠB - Technical University of Ostrava
17. listopadu 15, Ostrava, Czech Republic

Abstract

Data compression has been widely applied in many data processing areas. Compression methods use variable-length codes with the shorter codes assigned to symbols or groups of symbols that appear in the data frequently. There exist many coding algorithms, e.g. Elias-delta codes, Fibonacci codes and other variable-length codes which are often applied to encoding of numbers. Although we often do not consider time consumption of decompression as well as compression algorithms, there are cases where the decompression time is a critical issue. For example, a real-time compression of data structures, applied in the case of the physical implementation of database management systems, follows this issue. In this case, pages of a data structure are decompressed during every reading from a secondary storage into the main memory or items of a page are decompressed during every access to the page. Obviously, efficiency of a decompression algorithm is extremely important. Since fast decoding algorithms were not known until recently, variable-length codes have not been used in the data processing area. In this article, we introduce fast decoding algorithms for Elias-delta, Fibonacci of order 2 as well as Fibonacci of order 3 codes. We provide a theoretical background making these fast algorithms possible. Moreover, we introduce a new code, called the Elias-Fibonacci code, with a lower compression ratio than the Fibonacci of order 3 code for lower numbers; however, this new code provides a faster decoding time than other tested codes. Codes of Elias-Fibonacci are shorter than other compared codes for numbers longer than 26 bits. All these algorithms are suitable in the case of data processing tasks with special emphasis on the decompression time.

Keywords: data compression, fast decoding algorithms, Fibonacci codes, Elias-delta code, Elias-Fibonacci code

Email addresses: `jiri.walder@vsb.cz` (Jiří Walder), `michal.kratky@vsb.cz` (Michal Krátký), `radim.baca@vsb.cz` (Radim Bača), `jan.platos@vsb.cz` (Jan Platoš), `vaclav.snasel@vsb.cz` (Václav Snášel)

1. Introduction

Data compression has been widely applied in many data processing areas. Various compression algorithms were developed for processing text documents, images, video, etc. In particular, data compression is of the foremost importance and has been well researched as it is presented in excellent surveys [24, 31].

Various codes have been applied for data compression [25]. In contrast with fixed-length codes, statistical methods use variable-length codes, with the shorter codes assigned to symbols or groups of symbols that have a higher probability of occurrence. People who design and implement variable-length codes have to deal with these two problems: (1) assigning codes that can be decoded unambiguously and (2) assigning codes with the minimum average size.

In some applications, a prefix code is required to code a set of integers whose length is not known in advance. The prefix code is a variable-length code that satisfies the prefix attribute. As we know, the binary representation of integers does not satisfy this condition. In other words, the size n of the set of integers has to be known in advance for the binary representation since it determines the code size as $1 + \lfloor \log_2 n \rfloor$. Several prefix codes such as Elias [4], Fibonacci [6, 2], Golomb [8, 30], and Huffman codes [10] are well-known representatives of prefix codes.

Although we often do not consider time consumption of decompression as well as compression algorithms, there are cases where these times are a critical issue. Furthermore, there are applications where the time consumption of a decompression algorithm is more important than the time of a compression algorithm. For example, real-time compression of data structures [26, 7], wireless network communication [16], and text decompression [20, 5, 15, 18, 1, 19, 22, 27, 32] follow this issue. In the case of data structures, pages are decompressed during every reading from a secondary storage into the main memory or items of a page are decompressed during every access to the page. Obviously, efficiency of a decompression algorithm is extremely important.

Data structures (like B-tree [3] or R-tree [9]) often store similar items on one page. When difference coding [24] is applied to the items, it is necessary to compress small values. Variable-length codes are suitable for the compression of these values. Since fast decoding algorithms are not yet known and conventional decoding algorithms require long decoding times, variable-length codes have not been used in the compression of data structures, and, in general, in the data processing area.

The first effort of the fast decoding algorithm for Fibonacci codes of order ≥ 2 has been proposed in [11, 12]. Their mapping tables are large and are therefore not useful for large numbers. In contrast, our approach deals with the general length of numbers. Moreover, we introduce fast algorithms for several codes; therefore, the scalability of the proposed method is much higher.

In Section 2, theoretical issues of variable-length codes such as Elias-delta, Fibonacci of order 2 and 3 are described. Moreover, in this section, we introduce a new code called Elias-Fibonacci. In Section 3, we provide a theoretical background of the fast decoding algorithms. These algorithms are based on a finite

automaton. Since the number of automaton states is high, we introduce two types of automaton reduction in Section 4. In Section 5, we compare our work with other works. In Section 6, we describe the fast algorithms for Elias-delta code, Fibonacci codes of order 2 and 3 as well as the Elias-Fibonacci code. In Section 7, experimental results are presented and the proposed algorithms are compared to each other. In the last section, we conclude this paper and outline future works.

2. Overview of Universal Codes

In this section, we describe Elias-delta, Fibonacci family, and the new Elias-Fibonacci codes. We propose conventional coding/decoding algorithms for each code. Although there are other codes, like Elias-gamma [4] and Golomb codes [8, 30], in [30, 28] authors propose that Elias-delta and Fibonacci family codes provide better compression ratio than other codes. We also studied works [11, 12] where the Fibonacci of order 3 is recommended as the most effective code, in this case, for the compression of a textual data. The Huffman code is not suitable for our purpose because it requires the estimation of the probability distribution for all encoded numbers. Even if the probabilities are determined, each number receives a code which must be stored in a decoding table. Moreover, the length of the table is equal to the size of the number domain. Adaptive algorithms for Huffman [17] and Golomb [23] are also not suitable because our method requires fixed precomputed tables. A brief description of these codes can be seen on Wikipedia [29].

2.1. Elias-delta Code

The Elias-delta code is one of the most widely used prefix codes. This code has been introduced by Peter Elias [4]. In this code, each number is represented by a variable-length binary codeword. Some examples of coded numbers are shown in Table 1. An integer is coded as follows:

1. Let $B(n)$ be the binary representation of the number n without insignificant 0-bits. Let $B'(n)$ be $B(n)$ without the leading 1-bit.
2. Let $L(n)$ be the binary representation of the length (number of bits) of $B(n)$.
3. Let $Z(n)$ be a sequence of zeros, where the number of zeros is equal to the length of $L(n) - 1$.
4. The Elias-delta code is then the concatenation $E(n) = Z(n)L(n)B'(n)$.

In Algorithm 1, we see the conventional Elias-delta decoding algorithm. In this algorithm we use symbols |, &, <<, and >> for the bit OR, AND, left, and right shift operations, respectively. This algorithm includes three blocks related to the above depicted three parts of the Elias-delta code. The decoding process is as follows:

Table 1: Examples of Elias-delta codewords for some integers

n	$B(n)$	$Z(n)$	$L(n)$	$B'(n)$	$E(n) = Z(n)L(n)B'(n)$
		Length of $L(n)$ as the number of zeros−1	*Length of $B(n)$ as the binary value*	*$B(n)$ without the leading bit*	*Elias-delta code*
1	1	–	1	–	1
2	10	0	10	0	0 10 0
3	11	0	10	1	0 10 1
4	100	0	11	00	0 11 00
5	101	0	11	01	0 11 01
6	110	0	11	10	0 11 10
7	111	0	11	11	0 11 11
8	1000	00	100	000	00 100 000
⋮	⋮	⋮	⋮	⋮	⋮
100	1100100	00	111	100100	00 111 100100

1. Read zeros from an input bit stream. Read until the the first 1-bit is detected. Store the number of zeros into $Z(n)$. (Lines 4–8)
2. Store the most significant 1-bit into $L(n)$. Read $Z(n)$ bits from the stream into $L(n)$. (Lines 9–14)
3. Read $B'(n)$: meaning read $L(n) - 1$ bits from the input bit stream. $B(n)$ is built by adding the 1-bit as the most significant bit to $B'(n)$. (Lines 15-21)

2.2. Fibonacci Code

The Fibonacci code is based on Fibonacci numbers [14] and it was introduced in [6]. In [2], authors introduced the generalized Fibonacci code. Authors showed that the Elias-delta code is asymptotically longer than the generalized Fibonacci codes (of any order) for small integers, but becomes shorter at a certain point which depends on the order m of the generalized code.

Fibonacci numbers of order m ($m \geq 2$) are defined as follows:

$$F_i^{(m)} = F_{i-1}^{(m)} + F_{i-2}^{(m)} + \ldots + F_{i-m}^{(m)}, \text{ for } i \geq 1,$$

$$\text{where } F_{-m+1}^{(m)} = F_{-m+2}^{(m)} = \ldots = F_{-2}^{(m)} = 0,$$

$$\text{and } F_{-1}^{(m)} = F_0^{(m)} = 1.$$

In Table 2, we see examples of Fibonacci numbers of orders $m = 2$ and $m = 3$.

```
   input  : Elias-delta code bit stream
   output: Decoded number n
 1 Zn ← 0;
 2 Ln ← 0;
 3 n ← 0;
 4 bit ←stream.GetNextBit();
 5 while not bit do
 6     Zn ++;
 7     bit ←stream.GetNextBit();
 8 end
 9 Ln ← 1 <<Zn;
10 while Zn > 0 do
11     Zn −−;
12     bit ←stream.GetNextBit();
13     Ln ←Ln | bit << Zn;
14 end
15 Ln −−;
16 n ← 1 << Ln;
17 while Ln > 0 do
18     Ln −−;
19     bit ←stream.GetNextBit();
20     n ←n | bit << Ln;
21 end
```

Algorithm 1: Elias-delta decoding algorithm

Definition 1. *Fibonacci binary encoding and computation of its value*

Let $F^{(m)}(n) = a_0 a_1 a_2 \dots a_k$ be the Fibonacci binary encoding of a positive integer n with Fibonacci numbers of order m. The value of the Fibonacci binary encoding, denoted $V(F(n))$, is defined as follows

$$V(F(n)) = n = \sum_{i=0}^{k} a_i F_i^{(m)} \ (a_i \in \{0,1\}, 0 \leq i \leq k).$$

In the Fibonacci binary encoding, each bit represents a Fibonacci number $F_i^{(m)}$. Such a number has the property of not containing any sequence of m consecutive 1-bits [2]. This property is utilized for the construction of the Fibonacci code $\mathcal{F}^{(m)}(n)$ of number n. Fibonacci code $\mathcal{F}^{(m)}(n)$ maps n onto a binary string so that the string ends with a sequence of m consecutive 1-bits (denoted 1_m in the following text).

2.2.1. Fibonacci code of order 2

A positive integer n is encoded by the Fibonacci code of order 2 with the addition of 1-bit to $F^{(2)}(n)$. It can be done because two consecutive 1-bits do

Table 2: Examples of Fibonacci numbers of orders $m = 2$ and $m = 3$

i	$F_i^{(2)}$	$F_i^{(3)}$
-2	0	0
-1	1	1
0	1	1
1	2	2
2	3	4
3	5	7
4	8	13
5	13	24
6	21	44
7	34	81
8	55	149

not appear in $F^{(2)}(n)$. The Fibonacci of order 2 coding algorithm stores the bits in the reverse order; therefore, we utilize a *stack* for the storage.

Consequently, in the case of Fibonacci of order 2, the coding algorithm of a positive integer n is as follows:

1. For the positive integer n find the i-th index of the largest Fibonacci number satisfying $F_i^{(2)} \leq n$.
2. If $F_i^{(2)} \leq n$ compute $n = n - F_i^{(2)}$ and push the 1-bit to the *stack*. Otherwise push the 0-bit to the *stack*.
3. Set $i = i - 1$. If $i \geq 0$ go to Step 2.
4. While *stack* is not empty remove a *bit* from the *stack* and put *bit* at the end of $\mathcal{F}^{(2)}(n)$. Obviously, $\mathcal{F}^{(2)}(n)$ does not include any bit when this algorithm starts.
5. Put the 1-bit at the end of $\mathcal{F}^{(2)}(n)$.

In Table 3, we see examples of Fibonacci code of order 2 for some integers.

Example 1. *Let us consider* $n = 53$. *Due to the fact that* $F_7^{(2)} = 34 \leq 53 < F_8^{(2)} = 55$, *we set* $i = 7$. *Since* $F_7^{(2)} = 34 \leq 53$, *we push 1-bit to the stack (stack* $= 1$*). Then we compute* $n = 53 - 34 = 19$ *and* $i = i - 1 = 6$. *Since* $F_6^{(2)} = 21 > 19$, *we push 0-bit to the stack (*10*) and* $i = 5$. *Since* $F_5^{(2)} = 13 \leq 19$, *we push 1-bit to the stack (*101*),* $n = 19 - 13 = 6$ *and* $i = 4$. *Since* $F_4^{(2)} = 8 > 6$, *we push 0-bit to the stack (*1010*) and* $i = 3$. *Since* $F_3^{(2)} = 5 \leq 6$, *we push 1-bit to the stack (*10101*),* $n = 6 - 5 = 1$ *and* $i = 2$. *Since* $F_2^{(2)} = 3 > 1$, *we push 0-bit to the stack (*101010*) and* $i = 1$. *Since* $F_1^{(2)} = 2 > 1$, *we push 0-bit to the stack (*1010100*) and* $i = 0$. *Since* $F_0^{(2)} = 1 \leq 1$, *we push 1-bit to the stack (*10101001*),* $n = 0$, *and* $i = -1$. *Due to the fact that* $i < 0$, *we continue with Step 4. We pop bits from the stack and put them to* $\mathcal{F}^{(2)}(n)$. *This step reverses bits; therefore* $\mathcal{F}^{(2)}(n) = 10010101$. *Finally, we put 1-bit at the end of* $\mathcal{F}^{(2)}(n)$, *the result is as follows:* $\mathcal{F}^{(2)}(n) = 100101011$.

The decoding algorithm works in the opposite way (see Algorithm 2):

1. Set n, *prev*, and i to zero.
2. Read the current bit from the input.
3. If both current bit and *prev* are 1-bits then `END`.
4. If the current bit is 1-bit add the $F_i^{(2)}$ number to n.
5. Set $i = i + 1$ and *prev* to the current bit. Continue with Step 2.

Table 3: Examples of the Fibonacci code of order 2 for some integers

n	$\mathcal{F}^{(2)}(n)$
1	11
2	011
3	0011
4	1011
5	00011
6	10011
7	01011
8	000011
⋮	⋮
100	00101000011

input : stream coded by the Fibonacci code of order 2
output: Decoded number n

```
1  n ← 0;
2  prevBit ← 0;
3  pos ← 0;
4  while true do
5      bit ←stream.GetNextBit();
6      if prevBit and bit then break;
7      if bit then n ← n + F(2)_pos ;
8      pos ← pos +1;
9      prevBit ← bit;
10 end
```

Algorithm 2: Decoding algorithm for the Fibonacci code of order 2

2.2.2. Fibonacci code of order 3 *and higher*

It is not possible to create Fibonacci codes of order 3 and higher by only appending a sequence of $(m-1)$ 1-bits to $F^{(m)}(n)$. For example, let us suppose $F^{(3)}(5) = 101$ and $F^{(3)}(12) = 1011$. If we append the 11 sequence to

the Fibonacci binary encoding we obtain following codes: 10111 and 101111, respectively. We get a sequence of four 1-bits at the end of the second code. These two codes are indistinguishable during decoding because the code ends with three 1-bits. We receive the number 5 for both codes; however, the 1-bit at the end of the second code is taken as a part of the next number and lead to error.

Therefore, we must use the Fibonacci sum $S_n^{(m)}$ introduced in [2] to solve this problem. Authors also depicted a proof of the completeness of the code.

Definition 2. *Fibonacci sum*

Let $S_n^{(m)}$ be the sum of Fibonacci numbers. The sum is defined as follows:

$$S_n^{(m)} = \begin{cases} 0, & \text{for } n < -1 \\ \sum_{i=-1}^{n} F_i^{(m)}, & \text{for } n \geq -1 \end{cases}$$

Consequently, the coding algorithm for the Fibonacci code of order 3 and higher for any positive integer n is as follows:

1. If $n = 1$, then $\mathcal{F}^{(m)}(n) = 1_m$. `END`.
2. If $n = 2$, then $\mathcal{F}^{(m)}(n) = 01_m$. `END`.
3. Find k such that $S_{k-2}^{(m)} < n \leq S_{k-1}^{(m)}$. Let $Q = n - S_{k-2} - 1$.
4. Compute $F^{(m)}(Q)$.
5. Append 01_m as a suffix to $F^{(m)}(Q)$. If necessary, append leading 0-bits to make $\mathcal{F}^{(m)}(n)$ of length $m + k$.

Example 2. *Let us consider $n = 26$. In Step 3, we get $k = 5$, since $S_{5-2}^{(3)} < n \leq S_{5-1}^{(3)}$, $15 < 26 \leq 28$. We compute $Q = 26 - 15 - 1 = 10$. In Step 4, we get $F^{(3)}(10) = 1101$, since $10 = 1 + 2 + 7 = F_0^{(3)} + F_1^{(3)} + F_3^{(3)}$. Finally, we get $\mathcal{F}^{(3)}(26) = 10110111$ after appending the sequence 0111. Obviously, we use the reverse bit ordering in the $\mathcal{F}^{(3)}(26)$ code. Since the length is $m + k = 3 + 5 = 8$ we do not append any 0-bits.*

In Table 4, we see examples of the Fibonacci code of order 3 for some integers.

The decoding algorithm for the Fibonacci code of order 3 and higher for any positive integer n utilizes the queue holding the last $m + 1$ bits. The algorithm works as follows:

1. Set Q and i to 0.
2. Read a sequence of m consecutive bits from an input stream into the queue. If the sequence only includes 1-bits then $n = 1$. `END`.
3. Read the next bit from the stream into the queue. If the sequence is 01_m then $n = 2$. `END`.
4. Remove the bit from the queue. If the i-th bit is 1-bit then add the $F_i^{(m)}$ number to Q. Set $i = i + 1$. Read the next bit into the queue. Repeat this step until a sequence of 01_m is in the queue.

Table 4: Examples of the Fibonacci code of order 3 for some integers

n	$F(n)$	$k-1$	S_{k-1}	$F(Q)$	$\mathcal{F}^{(3)}(n)$
		Prefix length	*Fibonacci sum* $\sum_{i=-1}^{k-1} F_i$	*Encoded prefix* $Q = n - S_{k-1} - 1$	*Fibonacci of order m=3 code*
1	1	-	-	$F(0) =$ _	111
2	01	-	-	$F(0) =$ _	0111
3	11	1	2	$F(0) =$ _	0 0111
4	001	1	2	$F(1) = 1$	1 0111
5	101	2	4	$F(0) =$ _	00 0111
6	011	2	4	$F(1) = 1$	10 0111
7	0001	2	4	$F(2) = 01$	01 0111
8	1001	2	4	$F(3) = 11$	11 0111
⋮	⋮	⋮	⋮	⋮	⋮
100	10010110	7	96	$F(3) = 11$	1100000 0111

5. Calculate $n = Q + S_{i-1}^{(m)} + 1$.

We see the conventional Fibonacci of order 3 decoding in Algorithm 3. The queue of the length 4 is replaced by four variables for more efficient processing.

2.3. New Elias-Fibonacci code

In this section, we introduce a new Elias-Fibonacci code. In later sections, we will propose advantages of this code compared to other prefix codes. The Elias-Fibonacci code is a universal code for positive integers. It consists of two parts. The second part is a binary representation of the number n labeled $B(n)$. The first part is the length of $B(n)$, labeled $L(n)$, encoded by the Fibonacci code of order 2. We do not utilize a delimiter in this code; however, we utilize a sequence of two 1-bits between the end of $F^{(2)}(L(n))$ and start of $B(n)$. In other words, if we reach two 1-bits in a code, we read $L(n)$ and we know that we must read $L(n) - 1$ bits to complete $B(n)$.

The name of the new code originates from the Elias-gamma code [4]. The prefix of the Elias-gamma code is encoded by unary encoding. In the case of the new code, the prefix is encoded by the Fibonacci code of order 2. Therefore, we have chosen the name as a combination of the names Elias and Fibonacci.

The coding algorithm is defined as follows:

1. Let us suppose $B(n)$ of a number n and its length $L(n)$.
2. Compute $F^{(2)}(L(n))$.
3. The Elias-Fibonacci code is defined as the following concatenation:

$$EF(n) = F^{(2)}(L(n))B(n).$$

```
   input  : stream encoded by the Fibonacci code of order 3
   output: Decoded number n
 1 bit3 ←stream.GetNextBit();
 2 bit2 ←stream.GetNextBit();
 3 bit1 ←stream.GetNextBit();
 4 if bit1 and bit2 and bit3 then
 5     n ← 1;
 6     End;
 7 end
 8 bit0 ←stream.GetNextBit();
 9 if bit0 and bit1 and bit2 then
10     n ← 2;
11     End;
12 end
13 Q ← 0;
14 pos ← 0;
15 while not (bit0 and bit1 and bit2) do
16     if bit3 then  Q ← Q +F(3)pos;
17     pos ←pos + 1;
18     bit3 ←bit2;
19     bit2 ←bit1;
20     bit1 ←bit0;
21     bit0 ←stream.GetNextBit();
22 end
23 n ← Q +S(3)pos−1 + 1;
```

Algorithm 3: Decoding algorithm for the Fibonacci code of order 3

The decoding algorithm is defined as follows (see Algorithm 4):

1. Set $L(n)$ and i to 0.
2. Read one bit from the input stream. If the previous bit is the 1-bit, continue with Step 4. In the first step of the algorithm we suppose that the previous bit is the 0-bit.
3. Add the $F_i^{(2)}$ number to $L(n)$ in the case where the i-th bit is the 1-bit. Set $i = i + 1$. Continue with Step 2.
4. Add the 1-bit as the leading bit to $B(n)$.
5. Read remaining $L(n) - 1$ bits of $B(n)$ from the input stream.

3. General Principles of Fast Algorithms

In this section, we introduce general principles of fast algorithms. All fast algorithms are based on the finite automaton with precomputed mapping tables.

Table 5: Examples of the Elias-Fibonacci code for some integers

n	$B(n)$ *Binary code*	$L(n)$ *Length of* $B(n)$	$F^{(2)}(L(n))$	$EF(n)$ *Elias-Fibonacci code*
1	1	1	1	1 1
2	10	2	01	01 10
3	11	2	01	01 11
4	100	3	001	001 100
5	101	3	001	001 101
6	110	3	001	001 110
7	111	3	001	001 111
8	1000	4	101	101 1000
⋮	⋮	⋮	⋮	⋮
100	1100100	7	0101	0101 1100100

input : stream encoded by the Elias-Fibonacci code
output: Decoded number n

```
1  n ← 0;
2  len ← 0;
3  prevBit ← 0;
4  pos ← 0;
5  while true do
6      bit ←stream.GetNextBit();
7      if prevBit and bit then break;
8      if bit then len ← len +F(2)pos ;
9      pos ← pos +1;
10     prevBit ← bit;
11 end
12 len ← len −1 ;
13 n ←n | 1 << len;
14 while len > 0 do
15     len −−;
16     bit ←stream.GetNextBit();
17     n ←n | bit << len;
18 end
```

Algorithm 4: Decoding algorithm of the Elias-Fibonacci code

This automaton is described in Section 3.1. Section 3.2 describes an identification of automaton states by a brute-force algorithm. In Section 3.3, we explain mapping table building in more details. Since the number of automaton states is rather high, we propose two types of automaton reduction in Section 4.

3.1. Finite Automaton with Precomputed Mapping Tables

The basic idea of all proposed fast algorithms is to process an input stream byte by byte instead of bit by bit. As previously proposed, the algorithm is based on a finite automaton. Each state of the automaton represents the position in the corresponding bit-oriented algorithm described in Section 2. A precomputed mapping table is used for each state of the automaton.

The first step in the automaton construction is an identification of all its states. The second step in the proposed algorithm is to create a mapping table for an automaton state and the segment size. Let S denote the *segment size* and let N_{states} denote the number of the automaton states. Each segment of an input stream and state define the position of a result record in the mapping table. Obviously, mapping tables include $N_{states} * 2^S$ records.

The precomputed table included in each automaton state allows the conversion of segments of the input stream's bytes directly into decoded numbers. The mapping table also defines the new automaton state for each segment. The length of the mapping table exponentially increases with the segment size. In Section 7, we show that the proposed method can produce very good results even for small segment sizes like 1 B. The segment of 1 B in size has an advantage because it is handled quickly and it allows the mapping table to be a reasonable size.

In more details, each record of the mapping table (MAP) includes the following attributes:

- *Segment* – the bits of the input stream
- $NewState$ – the next state of the finite automaton
- $OutputCount$ – the amount of numbers which are decompressed from an actual segment. The maximum value is equal to $\left\lceil \frac{S-1}{L_{min}} \right\rceil + 1$, where L_{min} is the minimal length of a coded number. For example, $L_{min} = 2$ for the Fibonacci code of order 2, $L_{min} = 1$ for the Elias-delta code.
- $Numbers$ – the resulting decoded numbers
- U – the partially decoded number. Bits of the next segment(s) will complete the U number.
- L_U – the number of bits in the actual segment used to encode the U number
- *Rest* – the number of bits in next segments needed to read the complete U number

In Algorithm 5, the basic fast decoding algorithm based on the finite automaton is depicted. Some attributes of the mapping table described above are not used in this basic algorithm. These attributes are utilized in coding-specific fast algorithms proposed in Section 6. The algorithm works as follows:

1. Set the automaton to the initial state. (Line 1)
2. Read the current segment from the input stream. This algorithm is finished if all segments are read. (Lines 2–3)
3. Find the current record in the table: the record for the current segment and state of the automaton. (Line 4)
4. Output all numbers, meaning *Numbers*. (Line 5)
5. Change the state of the automaton to a new one defined in the current record, meaning the *NewState* attribute, and continue with Step 2. (Line 6)

```
   input  : stream  – coded bits
   output: result – array of decompressed numbers
1  state ← 0;
2  for x ← 0 to stream.GetByteCount() do
3      segment ← stream.GetNextByte();
4      map ← MAP[state,segment];
5      result ← result ⋃_{i=1}^{map.outputCount−1} map.Numbers[i];
6      state ←map.NewState;
7  end
```

Algorithm 5: Basic fast decoding algorithm based on the finite automaton

3.2. Identification of Automaton States with Brute-force Approach

There are various possibilities for identifying all automaton states. A brute-force algorithm is a simple solution; however, it leads to many states of the automaton. In the brute-force approach, we must remember the incomplete part of the code in the automaton state. Let us consider that each code is a sequence of 0- and 1-bits and the maximal length of a coded number is L_{max}. Each incomplete sequence of bits is of length $L_U < L_{max}$. Each L_U defines 2^{L_U} combinations of 0- and 1-bits. We obtain the total number of states for the brute-force approach when we summarize all of the incomplete reading possibilities plus one complete reading as follows:

$$N_{States} = \sum_{L_U < L_{\max}} 2^{L_U} + 1 = \sum_{x=1}^{L_{\max}-1} 2^x + 1 = 2\frac{2^{L_{\max}-1} - 1}{2 - 1} + 1 = 2^{L_{\max}} - 1$$

Example 3. *Let us consider that the maximal length of a coded number is* $L_{max} = 3$. *One complete reading and all incomplete readings of one bit and two bits are depicted in Figure 1. In this figure, the x symbol denotes an unknown bit. Each line in the figure corresponds to one automaton state; therefore, there is* $N_{states} = 2^{L_{\max}} - 1 = 2^3 - 1 = 7$ *states of the automaton.*

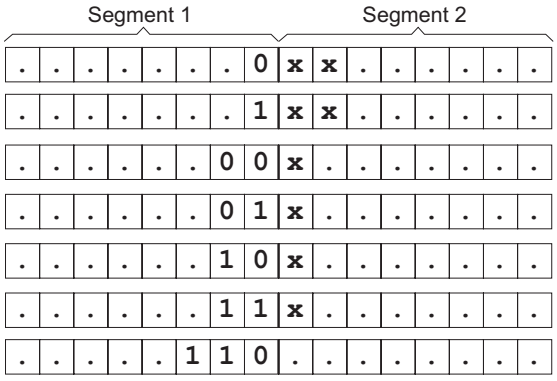


Figure 1: One complete reading and all incomplete readings for a code with the maximal length $L_{max} = 3$

3.3. Mapping Table Building

The algorithm for mapping table building is based on a conventional bit-oriented algorithm. During the building of the mapping table we must consider all states and possibilities when we read the current segment. The mapping table building algorithm is shown in Algorithm 6. This algorithm includes two loops; the first loop covers all states of the automaton and the second loop covers all possible combinations of the segment. We need to fill the appropriate mapping table line in each inner loop. We explain all functions called in the inner loop:

- *GetBitsForState* – returns bits which are known for the given state. When the conventional bit algorithm processes these bits, it waits for the next bit. Its local variables are set to values and the algorithm is positioned in a line. Consequently, the algorithm state corresponds to the automaton state.

  In the brute-force approach, these bits are set to the U number with the L_U length:

$$L_U = \lfloor \log_2(state + 1) \rfloor$$
$$U = state - 2^{L_U} + 1$$

  Example 4. *Let us consider* $state = 10$. *We obtain* $L_U = \lfloor \log_2(10 + 1) \rfloor \approx \lfloor 3.45 \rfloor = 3$ *and* $U = 10 - 2^3 + 1 = 3$. *The binary number 11 is increased to the length 3 by adding the 0-bit. Consequently, we obtain 011 bits for the state definition.*

  For $state = 0$ *we obtain* $L_U = U = 0$ *which corresponds to a complete reading which is the initial state of the automaton.*

For state $= 7$ *we obtain* $L_U = 3$ *and* $U = 0$. *In this case, we initialize the conventional algorithm with the sequence of 3 0-bits.*

- *GetBitsForSegment* – returns bits which are read from the actual segment. The *segment* value is filled by insignificant 0-bits to the length of the segment size.

  Example 5. *The segment* $= 34_{10} = 100010_2$ *value is filled to the length of the segment size* $S = 8$, *which means the result is 00100010.*

- *BitAlgorithm* – is any above described bit-oriented conventional algorithm. The algorithm processes *stream* filled by the *stream.SetBits* function. This algorithm assigns *OutputCount* and $Numbers$ parameters by decoded numbers. The algorithm ends when the *stream* is empty. Obviously, U and L_U variables contain unprocessed bits. These bits further define the next state of the automaton.

- *GetNewState* – this part finds the new state of the automaton from the U and L_U variables. If the new state is not found, it indicates an error in the input bit stream and further decoding is not possible. In this case, we assign the new state to -1.

  We can use a simple formula for the calculation of the new state in the brute-force approach:

$$NewState = \begin{cases} 2^{L_U} - 1 + U, & \text{for } L_U > L_{max} \\ -1, & \text{for } L_U \leq L_{max} \end{cases}$$

  Example 6. *Let us consider* $U = 010110 = 22_{10}$ *and* $L_U = 6$. *In the brute-force approach* $NewState = 2^6 - 1 + 22 = 85$.

```
   output: mapping table
1  for state ← 0 to N_states −1 do
2      for segment ← 0 to 2^S − 1 do
3          MAP[state,segment].Segment ← segment;
4          stream.SetBits(GetBitsForState(state)) ;
5          stream.SetBits(GetBitsForSegment(segment));
6          BitAlgorithm(stream,MAP[state,segment]);
7          MAP[state,segment].NewState ←
           GetNewState(MAP[state,segment]);
8      end
9  end
```

Algorithm 6: The mapping table building algorithm

Table 6: Result of the automaton reduction

	Elias-delta	Fibonacci $m = 2$	Fibonacci $m = 3$	Elias-Fibonacci
The code of $n = 15$	00100111	0100011	1100111	1011111
The length of code	8	7	7	7
The number of states for the Brute-force algorithm	255	127	127	127
The number of states after the reduction	**14**	**2**	**3**	**13**

4. Reduction of Automaton States Number

Since the number of automaton states is high, especially for longer segment sizes, we need to reduce the number of states. We apply two principles of automaton reduction. The first principle is the identification of similar states explained in Section 4.1. The second principle includes a shift operation introduced in Section 4.2.

4.1. Automaton States Similarity

When we analyze all automaton states created by the brute-force algorithm, we can identify some types of similar states. Moreover, similar states can be grouped into one state without an influence on the automaton functionality, and so the reduction is made in this way. In [11, 12], authors introduce a similar type of reduction for the Fibonacci; however, we introduce this reduction for more codes. We describe all individual states' similarities for each fast algorithm in the following sections. The automaton reduction has no influence on creation of the mapping table. The state loop in Algorithm 6 must be carried in the same way; however, the number of states is reduced. When the new state is not found by the *GetNewState* function it indicates an error in an input bit stream.

We show the result of this reduction in Table 6 for the 8-bit segment and 4-bit coded numbers. Since, we do not consider large numbers, the whole code is stored in one segment. In the case of the brute-force algorithm we also depict the maximal code and length of this code. The formulas for the calculation of the number of the automaton states are derived later for each fast algorithm.

4.2. Shift Operations

The shift operation is the second method of automaton reduction. It is necessary to utilize the shift operation for segmentation when the maximal length of a coded number is greater than the segment size, in other words: $L(n) > S$. In the case of Elias-delta as well as Elias-Fibonacci codes the shift operation is applied to the binary part of the code; therefore, we describe the binary shift operation in Section 4.2.1. On the other hand, the binary shift operation is not applicable for the Fibonacci code; therefore, we introduce a new Fibonacci shift operation in Section 4.3.

4.2.1. Binary Shift Operation

Let us consider the number n encoded by the binary code $B(n)$ where $L(n) > S$. It is not possible to completely decode the number, because it is separated into more than one segment. However, we can utilize a feature of the prefix codes: we know the length of each encoded number from the prefix of the code for Elias-delta as well as Elias-Fibonacci encodings.

If we use the binary encoding where the most significant bit is read first from an input bit stream, we can directly decode individual parts of the number. We can use the following procedure to decode the whole number:

1. Set $n = 0$ and $Rest = L(n)$
2. Decode the part of the number in the actual segment into P and set L_P to the number of bits read from the actual segment.
3. Compute $Rest = Rest - L_P$
4. Compute $n = n + P << Rest$
5. Finish this procedure if $Rest = 0$; otherwise continue with Step 2

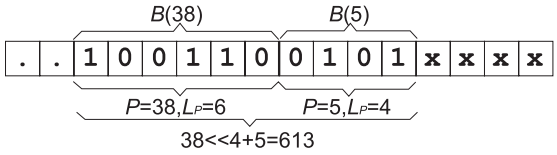


Figure 2: The shift operation of the encoded number stored in two segments

Example 7. *Let us consider the binary encoded number* $n = 613$. *The length of the binary code is 10 bits, we know the length* $Rest = L(n) = 10$ *in advance. The number is separated into two segments. This situation is depicted in Figure 2. When we read the first segment then* $P = 38$ *and* $L_P = 6$. *Since* $Rest = 10-6 = 4$ *then* $n = 0 + 38 << 4 = 608$*;* $Rest > 0$*; therefore, we continue with Step 2. When the algorithm finishes, we get* $P = 5$, $L_P = 4$, *and* $Rest = 4 - 4 = 0$. *We shift the number* $n = 608 + 5 << 0 = 613$. *Since* $Rest = 0$, *the number is completely decoded.*

In the brute-force approach, we must remember the partly decoded number in each state of the automaton. Otherwise, using the shift operation, we utilize the n variable containing the partially decoded number and *Rest* containing the length of the unread part of the number. When shifting, decoding the next segment depends on the *Rest* value and we must handle this situation by correctly setting the next automaton state. First, we number the automaton states by the *Rest* value. It means we use $state = 0$ for an initial state and state$\in < 1, S >$ in the case of incomplete number readings. If $Rest > S$ then the automaton remains in the current state, it means $state = S$. If we use the shift operation, we only need $S+1$ states instead of $2^{L_{\max}} - 1$ states used in the

Table 7: The number of automaton states for the 8 bit-length segment size and 32 and 64 bit-length encoded numbers after both reductions

Algorithm	N_{states}	
	32-bit	64-bit
Brute-force	$2^{32} - 1 = 4,294,967,295$	$2^{64} - 1$
Fast Elias-delta	$36 + 8 + 1 = 45$	78
Fast Elias-Fibonacci	$57 + 8 + 1 = 64$	151

brute-force approach. We compare these values in Table 7 for the 8-bit segment and 32 and 64 bit-length encoded numbers. We see that the number of states is dramatically reduced.

Let us have $S = 16$. In this case, $N_{states} = 53$ and 81 for Fast Elias-delta and Elias-Fibonacci codes, respectively. Moreover, we must store a table containing 2^{16} instead of 2^8 records for each state. We get the mapping tables with $53 \times 2^{16} = 3,473,408$ and $81 \times 2^{16} = 5,308,416$ records for $S = 16$ and 32 bit-length encoded numbers. Consequently, fast algorithms for $S = 8$ utilize the processor's L2 cache more efficient than fast algorithms using the 16 bit-length segment.

In Algorithm 7, we show a part of the shift operation algorithm; we show how to set the new state for the shift operation. Later in Sections 6.1 and 6.3 we will see how this operation is incorporated into Fast Elias-delta and Elias-Fibonacci algorithms.

```
1  n ← 0 ;
2  Rest ← L(n) ;
3  forall (P,L_P) do
4      state ←map.NewState;
5      if Rest ≥ S then
6          n ←n +(P <<Rest );
7          Rest ←Rest −L_P;
8          if Rest < S then
9              state ←Rest;
10         else
11             state ← S ;
12         end
13     end
14     n ←n +P;
15 end
```

Algorithm 7: Setting the new state for the shift operation

4.3. Fibonacci Shift Operation

Since we can not utilize the previously proposed binary shift for Fibonacci encoding, we introduce a special shift operation for this encoding: the Fibonacci shift. In the following, we write $F(n)$ for $F^{(m)}(n)$ when an arbitrary but fixed m is the underlying order of F_i.

Definition 3. *Fibonacci shift operation*

Let $F(n)$ be a Fibonacci binary encoding, k be an integer, $k \geq 0$. The k-th Fibonacci left shift $F(n) <<_F k$ is defined as follows:

$$F(n) <<_F k = \overbrace{00\ldots0}^{k} a_0 a_1 a_2 \ldots a_p$$

Fibonacci right shift is defined as is follows:

$$F(n) >>_F k = a_k a_{k+1} a_{k+2} \ldots a_p$$

It is easy to show that the Fibonacci shift is also the Fibonacci binary encoding; thus: $F(n) <<_F 0 = F(n) >>_F 0 = F(n)$.

For example, let us suppose the Fibonacci binary encoding of order $m = 2$, then $F(1) <<_F 2 = 1 <<_F 2 = 001 = F(3)$, $F(2) <<_F 3 = 01 <<_F 3 = 00001 = F(8)$ and $F(6) >>_F 3 = 1001 >>_F 3 = 1 = F(1)$.

The computation of $V(F(n) <<_F k)$ is done in the following steps:

1. Compute $F(n)$ for the n value according to Definition 1.
2. Shift bits of $F(n)$ by k: $F(n) << k$.
3. Compute the value from the shifted Fibonacci binary code $F(n) << k$ according to Definition 1.

The Fibonacci shift operation is time consuming since the Fibonacci value computation in Step 1 requires a summarization of Fibonacci numbers for 1-bits. In Step 2, we compute the binary shift (not Fibonacci shift) and finally, in Step 3, we compute the second summarization of Fibonacci numbers for assigned bits of an extended Fibonacci binary encoding.

To be able to compute a shifted value as fast as possible, we introduce a formula for the calculation of the Fibonacci left shift using Fibonacci numbers and Fibonacci right shift.

The formula is informally based on the following idea. We want to compute $V(0101)$ based on $V(101)$ for the Fibonacci binary encoding of order $m = 2$. $V(101) = F_0 + F_2$. $V(0101) = F_1 + F_3 = (F_{-1} + F_0) + (F_1 + F_2) = (F_0 + F_2) + (F_{-1} + F_1) = V(101) + V(01)$. It means $V(0101) = V(101) + V(01)$. Because $F(4) = 101$, we can set $V(0101) = F(4) <<_F 1$ and $V(01) = F(4) >>_F 1$. Formally, it can be written: $F(4) <<_F 1 = F(4) + (F(4) >>_F 1)$.

We prove the formula for the Fibonacci binary encoding of orders $m = 2$ and $m = 3$ in Theorems 1 and 2, respectively.

Theorem 1. *Calculation of the k-Fibonacci left shift of order $m = 2$*

Let $F^{(2)}(n)$ be the Fibonacci binary encoding of order $m = 2$ for the n value, then

$$V(F^{(2)}(n) <<_F k) = F^{(2)}_{k-1} \times V(F^{(2)}(n)) + F^{(2)}_{k-2} \times V(F^{(2)}(n) >>_F 1)$$

Proof 1. *In this proof, we use $F(n)$ instead of $F^{(2)}(n)$ and F_i instead of $F^{(2)}_i$. This theorem can be proved by mathematical induction. First, we show that the theorem holds for $k = 0$ and $k = 1$ (base case). Let us have $F(n) = a_0a_1a_2 \dots a_p$ then*

$$\begin{aligned} V(F(n) <<_F 0) &= F_{-1} \times V(F(n)) + F_{-2} \times V(F(n) >>_F 1) \\ &= 1 \times V(F(n)) + 0 \times V(F(n) >>_F 1) \\ &= V(F(n)) \end{aligned}$$

$$\begin{aligned} V(F(n) <<_F 1) &= \sum_{i=0}^{p} a_i F_{i+1} = a_0F_1 + a_1F_2 + \ldots + a_pF_{p+1} \\ &= a_0(F_0 + F_{-1}) + a_1(F_1 + F_0) + \ldots + a_p(F_p + F_{p-1}) \\ &= (a_0F_0 + a_1F_1 + \ldots + a_pF_p) + (a_0F_{-1} + a_1F_0 + \ldots + a_pF_{p-1}) \\ &= \sum_{i=0}^{p} a_iF_i + \sum_{i=0}^{p} a_iF_{i-1} \\ &= V(F(n)) + V(F(n) >>_F 1) \\ &= 1 \times V(F(n)) + 1 \times V(F(n) >>_F 1) \\ &= F_0 \times V(F(n)) + F_{-1} \times V(F(n) >>_F 1) \end{aligned}$$

By induction hypothesis, it is assumed that this theorem holds for all j, $0 \leq j < k$. We must prove the following equation:

$$V(F(n) <<_F k) = F_{k-1} \times V(F(n)) + F_{k-2} \times V(F(n) >>_F 1)$$

$$
\begin{aligned}
V(F(n) <<_F k) &= \sum_{i=0}^{p} a_i F_{k+i} \\
&= \sum_{i=0}^{p} a_i F_{k+i-1} + \sum_{i=1}^{p} a_i F_{k+i-2} \\
&= V(F(n) <<_F k-1) + V(F(n) <<_F k-2) \\
&= F_{k-2} \times V(F(n)) + F_{k-3} \times V(F(n) >>_F 1) + \\
&+ F_{k-3} \times V(F(n)) + F_{k-4} \times V(F(n) >>_F 1) \\
&= F_{k-2} \times V(F(n)) + F_{k-3} \times V(F(n)) + \\
&+ F_{k-3} \times V(F(n) >>_F 1) + F_{k-4} \times V(F(n) >>_F 1) \\
&= (F_{k-2} + F_{k-3}) \times V(F(n)) + (F_{k-3} + F_{k-4}) \times V(F(n) >>_F 1) \\
&= F_{k-1} \times V(F(n)) + F_{k-2} \times V(F(n) >>_F 1) \quad \blacksquare
\end{aligned}
$$

Theorem 2. *Calculation of the k-Fibonacci left shift of order $m = 3$*

Let $F^{(3)}(n)$ be the Fibonacci binary encoding of order $m = 3$ for the n value. Then

$$
\begin{aligned}
V(F^{(3)}(n) <<_F k) &= F^{(3)}_{k-1} \times V(F^{(3)}(n)) \\
&+ (F^{(3)}_{k-2} + F^{(3)}_{k-3}) \times V(F^{(3)}(n) >>_F 1) \\
&+ F^{(3)}_{k-2} \times V(F^{(3)}(n) >>_F 2)
\end{aligned}
$$

Proof 2. *In this proof, we use $F(n)$ for $F^{(3)}(n)$ and F_i for $F^{(3)}_i$. This theorem can be proved by mathematical induction as well. First, we show that the statement holds for $k = 0$, $k = 1$, and $k = 2$ (base case). Let us have $F(n) = a_0 a_1 a_2 \dots a_p$ then*

$$
\begin{aligned}
V(F(n) <<_F 0) &= F_{-1} \times V(F(n)) \\
&+ (F_{-2} + F_{-3}) \times V(F(n) >>_F 1) \\
&+ F_{-2} \times V(F(n) >>_F 2) \\
&= 1 \times V(F(n)) + (0+0) \times V(F(n) >>_F 1) + 0 \times V(F(n) >>_F 2) \\
&= V(F(n))
\end{aligned}
$$

$$
\begin{aligned}
V(F(n) <<_F 1) &= F_0 \times V(F(n)) \\
&+ (F_{-1} + F_{-2}) \times V(F(n) >>_F 1) \\
&+ F_{-1} \times V(F(n) >>_F 2) \\
&= 1 \times V(F(n)) + (1+0) \times V(F(n) >>_F 1) + 1 \times V(F(n) >>_F 2)
\end{aligned}
$$

$$\begin{aligned} V(F(n) <<_F 2) &= F_1 \times V(F(n)) \\ &+ (F_0 + F_{-1}) \times V(F(n) >>_F 1) \\ &+ F_0 \times V(F(n) >>_F 2) \\ &= 2 \times V(F(n)) + 2 \times V(F(n) >>_F 1) + 1 \times V(F(n) >>_F 2) \\ &= V(F(n) <<_F 1) + V(F(n)) + V(F(n) >>_F 1) \end{aligned}$$

By induction hypothesis, it is assumed that this theorem holds for all j, $0 \leq j < k$. *We must prove the following equation:*

$$\begin{aligned} V(F(n) <<_F k) &= F_{k-1} \times V(F(n)) \\ &+ (F_{k-2} + F_{k-3}) \times V(F(n) >>_F 1) \\ &+ F_{k-2} \times V(F(n) >>_F 2) \end{aligned}$$

$$\begin{aligned} V(F(n) <<_F k) &= \sum_{i=0}^{p} a_i F_{k+i} \\ &= \sum_{i=0}^{p} a_i F_{k+i-1} + \sum_{i=1}^{p} a_i F_{k+i-2} + \sum_{i=1}^{p} a_i F_{k+i-3} \\ &= V(F(n) <<_F k-1) + V(F(n) <<_F k-2) + V(F(n) <<_F k-3) \\ &= F_{k-2} \times V(F(n)) \\ &+ (F_{k-3} + F_{k-4}) \times V(F(n) >>_F 1) \\ &+ F_{k-3} \times V(F(n) >>_F 2) \\ &+ F_{k-3} \times V(F(n)) \\ &+ (F_{k-4} + F_{k-5}) \times V(F(n) >>_F 1) \\ &+ F_{k-4} \times V(F(n) >>_F 2) \\ &+ F_{k-4} \times V(F(n)) \\ &+ (F_{k-5} + F_{k-6}) \times V(F(n) >>_F 1) \\ &+ F_{k-5} \times V(F(n) >>_F 2) \\ &= (F_{k-2} + F_{k-3} + F_{k-4}) \times V(F(n)) \\ &+ (F_{k-3} + 2 \times F_{k-4} + 2 \times F_{k-5} + F_{k-6}) \times V(F(n) >>_F 1) \\ &+ (F_{k-3} + F_{k-4} + F_{k-5}) \times V(F(n) >>_F 2) \\ &= F_{k-1} \times V(F(n)) \\ &+ (F_{k-2} + F_{k-3}) \times V(F(n) >>_F 1) \\ &+ F_{k-2} \times V(F(n) >>_F 2) \quad \blacksquare \end{aligned}$$

The Fibonacci shift manipulates Fibonacci binary encoded numbers in the same way as the shift operation manipulates binary numbers. Because the shift is applied differently we do not know the length of a coded number in advance, and therefore, we utilize the delimiter. The delimiter is the last part of all Fibonacci codes as was described in Sections 2.2.1 and 2.2.2. The Fibonacci binary encoded number is always located before the delimiter. The most significant bit is read last from a bit stream. The Fibonacci shift is carried out by applying the following procedure (see Algorithm 8):

1. Set $n = 0$ and $len = 0$
2. Decode the part of the number in the actual segment into P and set L_P to the number of incomplete bits decoded from the actual segment.
3. Calculate $n = n + P <<_F len$
4. Calculate $len = len + L_P$
5. When the delimiter is not read from the actual segment continue with step 2.

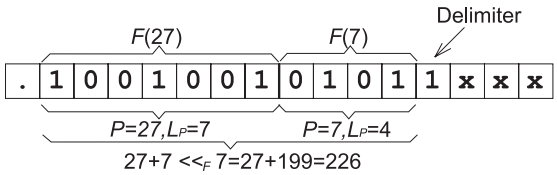


Figure 3: The shift of the Fibonacci code of order $m = 2$ between segments

Example 8. *Let us consider the binary encoded number* $n = 226$. *The length of the Fibonacci code of order* $m = 2$ *for this number is 11 bits plus one bit for the delimiter. The number is separated into two segments. This situation is depicted in Figure 3. When we read the first segment then* $U = 27$ *and* $L_U = 7$. *Therefore,* $n = 0 + 27 <<_F 0 = 27$ *and* $len = 7$. *Since the delimiter has not been reached we continue with Step 2. After reading the next segment we obtain* $U = 7$ *and* $L_U = 4$ *and we calculate* $n = 27 + 7 <<_F 7 = 27 + V(0101) <<_F 7 = 27 + V(0101) * F_6 + F_5 * V(0101 >>_F 1) = 27 + 21 * 7 + 13 * 4 = 27 + 199 = 226$ *and* $len = 11$. *The Fibonacci shift in the formula is calculated according to Theorem 1. Since the delimiter has been found the number was completely decoded.*

In the case of the previous fast decoding algorithms, we must remember a partially decoded number in automaton states. Obviously, these numbers define the next automaton states. On the other hand, in the case of the Fibonacci encoding, we can remember a partially decoded number in the n variable; see Algorithm 8 for more details.

```
1  n ← 0 ;
2  len ← 0 ;
3  forall (P,L_P) do
4      state ←map.NewState;
5      if len > 0 then
6          n ← n + V(F(P <<_F len));
7      else
8          n ← P;
9      end
10     len ← len +L_P;
11 end
```

Algorithm 8: A part of the fast algorithm for the Fibonacci shift operation

In Algorithm 8, the shift is only carried out for numbers which fit into one segment. According to Theorems 1 and 2 we can calculate these shifts effectively with a utilization of precomputed Fibonacci right shifts. We need to precompute the 1-Fibonacci right shift for order $m = 2$ and 1- and 2-Fibonacci right shift for order $m = 3$.

Example 9. *Let us consider the Fibonacci encoding of order* $m = 2$ *and* $n = 32$. *The Fibonacci binary encoding* $F(n) = 0010101$ *and the precomputed right shift is* $V(F(n) >>_F 1) = V(010101) = 20$. *The value of the 3-Fibonacci left shift is calculated as follows:*

$$\begin{aligned} V(F(n) <<_F k) &= F_{k-1} \times V(F(n)) + F_{k-2} \times V(F(n) >>_F 1) \\ V(0010101 <<_F 3) &= F_{3-1} \times 32 + F_{3-2} \times 20 \\ &= F_2 \times 32 + F_1 \times 20 \\ &= 3 \times 32 + 2 \times 20 \\ &= 96 + 40 = 136 \\ &= V(0000010101) \end{aligned}$$

5. Comparison with Other Works

A fast algorithm for the Fibonacci code proposed by Shmuel T. Klein in [11, 12, 13] also applies a kind of mapping table for each decoded segment and it utilizes an automaton reduction by the state similarity as well. Author proposed two approaches for the computation of the Fibonacci shift. In the first approach, the Fibonacci shift is calculated utilizing properties of the Fibonacci code. Author utilizes float numbers for the computation and this issue results in a slower computation. Therefore, the second approach gets rid of float numbers. In this case, the Fibonacci shift is precomputed for each k-shift. It means the mapping table length depends on the code length as well as the segment size. For $S = 8$ and the coded numbers of maximum values 10,000 (14 bit-length numbers), 100,000 (17 bit-length numbers), and 1,000,000 (20 bit-length

numbers), the table lengths are 10k, 15k, and 21.4k, respectively. For $S = 16$ the table lengths are 1.9M, 2.6M, and 3.4M, respectively. On the other hand, our Fast Fibonacci algorithm is not affected by the length of the coded numbers. For $S = 8$, the table lengths are $2 \times 2^8 = 512$ and $3 \times 2^8 = 768$ for order $m = 2$ and $m = 3$, respectively. For $S = 16$, the mapping table lengths are $2 \times 2^{16} = 131,072$ and $3 \times 2^{16} = 196,608$ for order $m = 2$ and $m = 3$, respectively. Although the second approach provides the same time complexity as our implementation, the mapping table length is bigger. Author utilizes the method for text compression; therefore, the mapping table length is sufficient for lower bit-lengths of coded numbers. However, our approach provides better scalability, due to the fact the table length is smaller even for higher coded numbers. Moreover, we introduce the fast decoding algorithms for other codes like Elias-delta and Elias-Fibonacci.

6. Fast Decoding Algorithms

6.1. Fast Elias-delta Decoding

In this case, the number of automaton states depends on the bit length of coded numbers. Let $L(n)$ denote the bit length of a coded number n and $Z(n)$ the length of $L(n)$ -1, which means $Z(n) = \lfloor \log_2 (B_{MAX}) \rfloor$, where B_{MAX} is the length of the maximum encoded number. If we analyze the Elias-delta code, we identify the following types of automaton states:

- *Initial state* – it represents the first state of the automaton where we read bits from the beginning of a coded number. This state arises when a coded number ends in the end of the actual segment.
- *Zero states* – it represents the reading of 0-bits. The number of zeros is equal to $Z(n)$; it defines $Z(n)$ states of the automaton. The number of states for a B_{MAX} value is equal to $Z(n) = \lfloor \log_2 (B_{MAX}) \rfloor$. Clearly, we do not know the number of zeros in the next segment when reading the current segment.

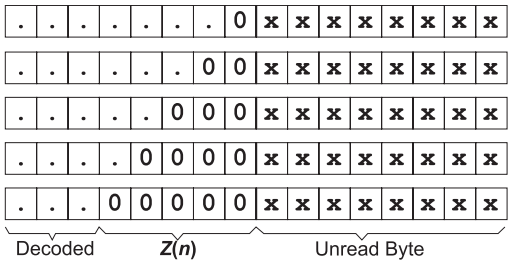


Figure 4: An example of partially decoded zeros where $L(n) = 32_{10} = 100000_2$

Example 10. *Let us consider an example where* $L(n) = 32_{10} = 100000_2$. *We need* $Z(n) = \lfloor \log_2(32) \rfloor = 5$ *zeros for coding the length of* $L(n)$. *In Figure 4, all combinations of the number of zeros in a segment are shown. Let us consider the third line of the figure; 3 bits of* $Z(n)$ *are included in the current segment; it means that the last 2 bits are included in the next segment.*

- *Length states* – it represents the reading of the $L(n)$ value. If we only consider the lengths of power of 2, it means $L(n) \leq 2^{Z(n)}$, the number of states is equal to $L(n) - 1$. This value is computed by considering two situations:

  1. $L(n) = 2^{Z(n)}$ (the longest number) – the number of states is equal to $Z(n)$.

  Example 11. *Let us consider an example where* $L(n) = 32_{10} = 100000_2 = 2^5$, $Z(n) = 5$. *We can get the following 5 combinations during the reading of* $L(n)$: *0000010000X, 000001000XX, 00000100XXX, 0000010XXXX, and 000001XXXXX.*

  2. $L(n) < 2^{Z(n)}$ – the number of states is equal to $2^{Z(n)} - 1 - Z(n)$. Here, we must summarize combinations for all $z < Z(n)$. The summarization is as follows

$$\sum_{x=1}^{z<Z(n)} (2^x - 1) = \sum_{x=1}^{Z(n)-1} (2^x - 1) = \sum_{x=1}^{Z(n)-1} 2^x - (Z(n) - 1)$$
$$= 2\frac{2^{Z(n)-1} - 1}{2 - 1} - (Z(n) - 1) = 2^{Z(n)} - 1 - Z(n)$$

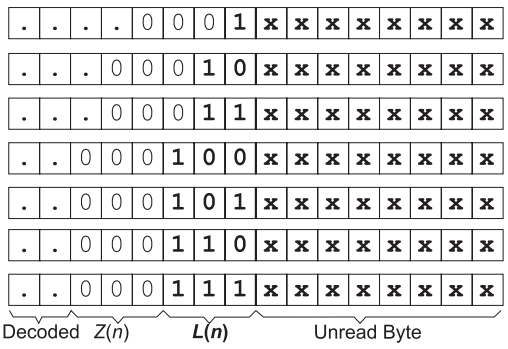


Figure 5: Partially decoded length where $Z(n) = 3$

Example 12. *Let us consider all zeros* $z < Z(n) = 5$ *for coding all lengths* $L(n) < 32$. *To obtain all combinations we need to summarize all sub-combinations for* $z \in \{1, 2, 3, 4\}$. *Therefore, we compute* $2^4 - 1 = 15$ *and* $2^3 - 1 = 7$ *combinations for* $z = 4$ *and* $z = 3$, *respectively. All combinations for* $z = 3$ *are depicted in Figure 5. Similarly, we get 3 combinations for* $z = 2$, *and 1 combination for* $z = 1$. *Finally, we summarize all computed values and the result is* $1 + 3 + 7 + 15 = 26$ *combinations. The same value is computed by* $2^{Z(n)} - 1 - Z(n) = 2^5 - 1 - 5 = 26$.

Finally, the whole number of the length states for $L(n) = B_{MAX}$ is as follows:

$$Z(n) + 2^{Z(n)} - 1 - Z(n) = 2^{Z(n)} - 1 = L(n) - 1 = B_{MAX} - 1$$

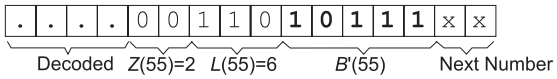


Figure 6: An example of a completely decoded number where $Rest = 0$

- *Binary number states* – the last part is the reading of the binary encoded number. The number of these states is equal to S (the segment size), since we utilize the shift operation (see Section 4.2). The shift operation is not applied when the binary number part is stored in one segment. This scenario is shown in Figure 6. On the other hand, if the binary part is divided into more than one segment, it means $Rest > 0$, and we need to shift the currently decoded part by $Rest$. This scenario is shown in Figure 7.

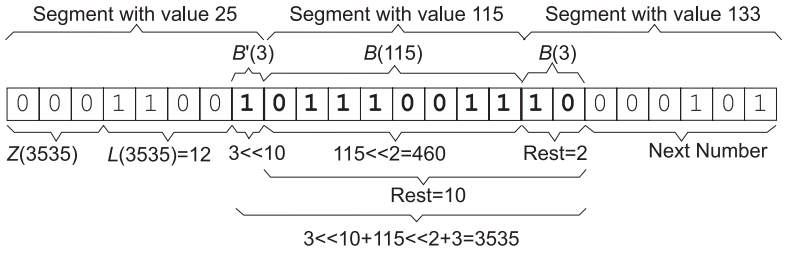


Figure 7: An example of a partially decoded number where $Rest > 0$. The partially decoded number 3 must be shifted by 10 and partially decoded number 115 must be shifted by 2.

In summary, the total number of states and the mapping table length are computed as follows:

$$\begin{aligned} N_{states} &= (1 + \lfloor \log_2 (B_{MAX}) \rfloor + B_{MAX} - 1 + S) \\ Length\_of\_table &= (1 + \lfloor \log_2 (B_{MAX}) \rfloor + B_{MAX} - 1 + S) \times 2^S \end{aligned}$$

```
   input  : stream coded by the Elias-delta code
   output: Array of decoded numbers result
 1 n ← 0;
 2 state ← 0;
 3 pos ← 0;
 4 rest ← 0;
 5 for x ← 0 to stream.GetByteCount() do
 6     if state < 0 then break;
 7     byte ← stream.GetNextByte();
 8     map ← MAP[state,segment];
 9     state ←map.NewState;
10     if rest ≥ S then
11         rest ←rest − S;
12         n ←n +(byte <<rest );
13         if rest = 0 then
14             result ← result ∪ n;
15             n ← 0;
16         end
17         if rest < S then state ←rest;
18     else
19         rest ←map .Rest;
20         if map.OutputCount > 0 then
21             n ←map.Numbers [0]+n;
22             result ← result ∪ n;
23             result ← result ⋃_{i=1}^{map.OutputCount−1} map.Numbers[i];
24         end
25         n ←map.U <<rest;
26     end
27 end
```

Algorithm 9: Fast Elias-delta decoding algorithm

The Fast Elias-delta decoding algorithm is shown in Algorithm 9. The algorithm utilizes the shift operation when it reads the binary part of a code. The shift operation is applied in two parts of the algorithm. If $Rest \geq S$ (Lines 13–21), we also need to handle the new automaton state. If $Rest < S$ (Lines 23–29), the reading of the binary part is finished. We start reading the binary part in Line 29.

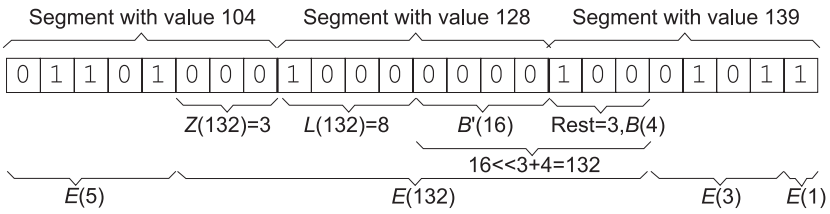


Figure 8: An example of the Fast Elias-delta decoding

Example 13. *Let us consider an example of the Fast Elias-delta decoding in Figure 8. We access the following records in the mapping table of the automaton:*

$$MAP[Segment = 104, State = 0] = \{NewState = 23, Rest = 0, U = 0, OutputCount = 1, Numbers = \{5\}\}$$
$$MAP[Segment = 128, State = 23] = \{NewState = 3, Rest = 3, U = 16, OutputCount = 0, Numbers = \{\}\}$$
$$MAP[Segment = 139, State = 3] = \{NewState = 0, Rest = 0, U = 0, OutputCount = 3, Numbers = \{4, 3, 1\}\}$$

After reading the first segment, we get the number 5 *directly from the mapping table (see* $Numbers$*); it is stored in the result array. The automaton is currently in the state 23. Clearly, we have three 0-bits of* $Z(n)$ *in the first segment. Now, the second segment with the value 128 is read. In the mapping table, we see that it represents the incomplete number* 16*. This value is stored in the* n *variable. Since there are* 3 *remaining bits (see the Rest value), the number* 16 *is right shifted by* 3 *bits; the result is* $n = 128$*. The next segment with the value 139 starts with the remaining 3 bits; the sequence* 100 *is read. We get the number* 4 *(see* $Numbers$*) which is added to* n*:* $n = 128 + 4 = 132$*. This value is stored in the result array with other two numbers* 3 *and* 1 *of* $Numbers$*.*

6.2. Fast Fibonacci Decoding

The structure of the Fast Fibonacci algorithm is the same for any order m of the Fibonacci code. The Fast Fibonacci decoding algorithm is shown in Algorithm 10. This algorithm differs for various orders in the following parts:

- *Mapping table*: The mapping table is based on a different bit-oriented algorithm. Utilization of the mapping table is located in Line 9.
- *Fibonacci left shift operation*: The Fibonacci shift operations are different for different orders (see Section 4.3). This operation is utilized in Lines 13 and 25 of Algorithm 10.

- *Final adjustment*: In the case of Fibonacci codes of order $m > 2$ we need to add $S_n + 1$. The algorithm handles this different computation in Lines 17 – 19.

```
   input  : stream encoded by the Fibonacci code of order m and the
            Fibonacci order m
   output: Array of decoded numbers result
 1 n ← 0;
 2 state ← 0;
 3 len ← 0;
 4 for x ← 0 to stream.GetByteCount() do
 5     if state < 0 then break;
 6     byte ← stream.GetNextByte();
 7     map ← MAP[state,segment];
 8     state ←map.NewState;
 9     for i ← 0 to map.OutputCount-1 do
10         if len > 0 then
11             n ← n + V(F(map.Numbers[i] <<_F len));
12         else
13             n ← map.Numbers[i];
14         end
15         if m > 2 then
16             n ←n +S_len + 1;
17         end
18         result ← result ∪ n;
19         n ← 0;
20         len ← 0;
21         continue;
22     end
23     if map.L_U > 0 then
24         n ← n + V(F(map.U <<_F len));
25         len ← len +map.L_U;
26     end
27 end
```

Algorithm 10: Fast Fibonacci decoding algorithm

The number of automaton states depends on the code order m: the automaton includes exactly m states. The number of states is derived from the idea that the 1-bits read from the start of the segment can complete the compressed number from a previous segment. An example of the Fibonacci code of order $m = 2$ is shown in Figure 9. The first bit in the segment 165 completes the decoded number 7. Due to this fact, it is necessary to have m states including the number of 1-bits in the end of the previous segment. The mapping table for

the automaton is precomputed by a conventional bit-oriented algorithm as was explained in Section 3.3.

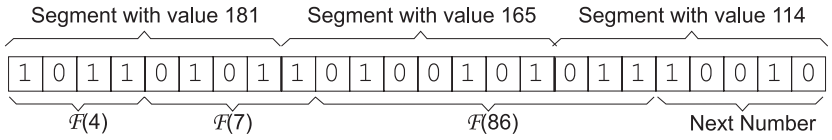


Figure 9: An example of compressed segments where the second segment completes the first one. The second segment defines another state of the automaton.

The previously introduced Fibonacci shift operation is utilized in two parts of the algorithm. When the number is not completely decoded the algorithm shifts the number in Lines 24–27. When the delimiter is found the number is completed; therefore, the shift operation is carried out inside the loop in Lines 11–23. The shift operation is located in Line 13.

The mapping table length for the Fibonacci code of order m is as follows:

$$\begin{aligned} N_{states} &= m \\ Length\_of\_table &= m \times 2^S \end{aligned}$$

Example 14. *Let us consider the Fibonacci code of order* 2 *and Figure 9. We access the following records of the mapping table during the decoding:*

$$\begin{aligned} &MAP[State = 0, Segment = 181] = \\ &\quad \{NewState = 1, U = 7, L_U = 4, OutputCount = 1, Numbers = \{4\}\} \\ &MAP[State = 1, Segment = 165] = \\ &\quad \{NewState = 1, U = 31, L_U = 7, OutputCount = 1, Numbers = \{0\}\} \\ &MAP[State = 1, Segment = 114] = \\ &\quad \{NewState = 0, U = 6, L_U = 5, OutputCount = 1, Numbers = \{2\}\} \end{aligned}$$

One number, the number 4, is obtained from the first record in the $Numbers$ *array. The number* 4 *is immediately stored in the result array. Incomplete number* $U = 7$ *is stored in the* n *variable holding each incomplete number from the previous segment. We set the number of incomplete bits* $len = L_U = 4$*. The segment ends with the 1-bit, and therefore we continue with the state 1 of the automaton and we read the second segment. Since it starts with the* 1*-bit, it completes the number stored in the* n *variable with 0 (see* $Numbers$*) and the result value is shifted by the len value; meaning* $n = n+U <<_F len = 4+0 <<_F 4 = 4+0 <<_F 4 = 4$*. The number 4 is stored in result and we set* $n = U = 31$ *and* $len = L_U = 7$*. The next segment starts with the sequence* $011 = F(2)$*; it completes the previous segment:* $n = n + U <<_F len = 31 + V(F(2) <<_F 7) = 31 + 55 = 86$*. The number 86 is stored in result and we set* $n = U = 6$ *and* $len = L_U = 5$*. The* n *value will be completed in the next segment.*

6.3. Fast Elias-Fibonacci Decoding

The Fast Elias-Fibonacci decoding algorithm is the same as the Fast Elias-delta decoding algorithm (see Algorithm 9); however, the mapping tables are different since automatons states, and of course conventional bit-oriented algorithms, are different.

In this case, the number of states only depends on the segment size S and the length of the maximum encoded number B_{MAX}. To derive a formula for the calculation of the automation states number we need to prove Theorem 5. In this proof, we apply Theorems 3 and 4 introduced and proved in [2].

Theorem 3. *Simplified Fibonacci sum computation*

The sum of Fibonacci numbers $S_i^{(m)}$ *is computed as follows:*

$$S_i = \frac{1}{m-1}(F_{n+2} - 1 + \sum_{i=0}^{m-3}(m-2-i)F_{n-i})$$

Theorem 4. *Length of Fibonacci binary code*

Let $S_i^{(m)}$ *be the sum of Fibonacci numbers; then all values* $V(F(n)) \in [S_{i-1}+1, S_i]$ *have the bit-length* $L = m + i$.

Theorem 5. *Interval of* $V(F(n))$ *for* $L(n)$

Let $L(n)$ *be the length of* $F(n)$ *of order* $m = 2$ *then all values* $V(F(n)) \in \left[F_{L(n)-1}, F_{L(n)} - 1\right]$ *have the length* $L(n)$.

Proof 3. *We set* $L = m + i$, $i = L - m$ *and we change indices in intervals proposed in Theorem 4 to* $\left[S_{L(n)-m-1} + 1, S_{L(n)-m}\right]$. *The interval is* $\left[S_{L(n)-2-1} + 1, S_{L(n)-2}\right]$ *for* $m = 2$. *After applying Theorem 3 for* $m = 2$, *we obtain* $S_i = F_{i+2} - 1$. *After this substitution, we obtain the result interval* $\left[F_{L(n)-1}, F_{L(n)} - 1\right]$.

Example 15. *Let us consider all Fibonacci binary codes of order 2 with the length* $L = 6$. *The interval bounds are* $S_3 = F_5 - 1 = 1 + 1 + 2 + 3 + 5 = 12$ *and* $S_4 = F_6 - 1 = S_3 + 8 = 20$. *Therefore, the interval of numbers is* $I_4 = [13, 20]$. *It means that the Fibonacci binary code for integers in the interval* $13 - 20$ *has the length* $L = 6$.

Automaton state types are as follows:

- *Initial state* – it represents the reading from the beginning of a coded number. This type defines the first automaton state. The state is set when the coded number ends in the end of the actual byte.
- *Fibonacci prefix states* – it represents the reading of the prefix in Fibonacci binary encoding. The length of the prefix is derived from the largest encoded number. We apply the Fibonacci binary encoding on B_{MAX}, we get $F(B_{MAX})$. For the sake of simplicity, let us use the abbreviation L_{FB} for $L(F(B_{MAX})$ in the following text. According to Theorem 5, the value

$B_{MAX} = V(F(B_{MAX})) \in [F_{L_{FB}-1}, F_{L_{FB}} - 1]$. If the prefix has been read in the current segment, we must decide if the prefix is complete or if it must be completed in the next segment:

1. *Incomplete*
   All incomplete prefixes are computed by a summarization of all possible prefix combinations and the number of these combinations is limited by a lower boundary of the above proposed interval. The summarization is as follows:

$$\sum_{L<L_{FB}} F_L = \sum_{x=1}^{L_{FB}-1} F_x = S_{L_{FB}-1} - 2 = F_{L_{FB}+1} - 3$$

2. *Complete*
   Completed prefixes are all of the length L_{FB}. There are at least $B_{MAX} + 1$ complete prefixes (0,1,2,...,B_{MAX}) encoded by Fibonacci binary encoding. Not all of these prefixes are used, since prefixes shorter than L_{FB} are already included in the previous case; therefore, the number of these prefixes is subtracted from $B_{MAX} + 1$. There are $F_{L_{FB}-1}$ Fibonacci codes shorter than L_{FB}.

If we combine these values, we get the following number of states:

$$F_{L_{FB}+1} - 3 + B_{MAX} + 1 - F_{L_{FB}-1}$$

Example 16. *If we want to encode the prefix for 16 bit-length numbers, the number of states is 35. Table 8 includes an example of state combinations for various bit-length numbers. We compute* $F(16) = 001001$ *and* $L_{FB} = L(F(B_{MAX})) = 6$. *Prefixes 1 – 31 define states for incomplete prefixes. Their lengths are shorter than 6 bits. If we summarize these states we get the following number of the incomplete prefixes:* $F_{L_{FB}+1} - 3 = F_{6+1} - 3 = 31$. *There are at least* $B_{MAX} + 1 = 16 + 1 = 17$ *combinations which can complete these prefixes; however, we do not compute* $F_{L_{FB}-1} = F_{6-1} = 13$ *combinations, due to the fact that the Fibonacci binary code shorter than* $L_{FB} = 6$ *is defined for values 0–12. These codes are also included in the incomplete prefixes. Consequently, the final number of states is equal to* $31 + 17 - 13 = 35$.

We get 64 states for 32-bit numbers, because $F(32) = 0010101$, $L_{FB} = L(F(B_{MAX})) = 7$, *and all prefix combinations are* $F_{7+1} - 3 + B_{MAX} + 1 - F_{7-1} = 55 - 3 + 32 + 1 - 21 = 64$.

Table 8: An example of all states to encode prefixes for 16 bit-length numbers

Prefix order	Prefix	Bits
1	XXXXXXX0	0
2	XXXXXXX1	1
3	XXXXXX00	0
4	XXXXXX10	1
5	XXXXXX01	2
6	XXXXX000	0
7	XXXXX100	1
8	XXXXX010	2
9	XXXXX001	3
10	XXXXX101	4
11	XXXX0000	0
12	XXXX1000	1
13	XXXX0100	2
14	XXXX0010	3
15	XXXX1010	4
16	XXXX0001	5
17	XXXX1001	6
18	XXXX0101	7
19	XXX00000	0
20	XXX10000	1
21	XXX01000	2
22	XXX00100	3
23	XXX10100	4
24	XXX00010	5
25	XXX10010	6
26	XXX01010	7
27	XXX00001	8
28	XXX10001	9
29	XXX01001	10
30	XXX00101	11
31	XXX10101	12
-	~~XX000000~~	0
-	~~XX100000~~	1
-	~~XX010000~~	2
-	~~XX001000~~	3
-	~~XX101000~~	4
-	~~XX000100~~	5
-	~~XX100100~~	6
-	~~XX010100~~	7
-	~~XX000010~~	8
-	~~XX100010~~	9
-	~~XX010010~~	10
-	~~XX001010~~	11
-	~~XX101010~~	12
32	XX000001	13
33	XX100001	14
34	XX010001	15
35	**XX001001**	16

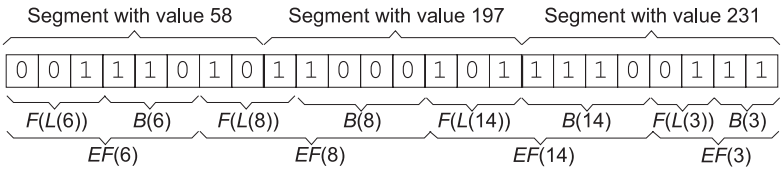


Figure 10: An example of a directly decoded number; the shift operation is not applied

- *Binary number states* – the last part is the reading of the binary encoded number. The number of these states is equal to the length of the segment size S because we apply the shift operation. This operation is not applied when the binary number part of the Elias-Fibonacci code is within one segment. This scenario is shown in Figure 10. On the other hand, if the binary part is spread to more segments, it means $Rest > 0$, and we need to shift the currently decoded part by $Rest$. This scenario is shown in Figure 11.

The number of automaton states and the length of the mapping table are as follows:

$$\begin{aligned} N_{states} &= 1 + F_{L_{FB}+1} - 3 + B_{MAX} + 1 - F_{L_{FB}-1} + S \\ Length\_of\_table &= (1 + F_{L_{FB}+1} - 3 + B_{MAX} + 1 - F_{L_{FB}-1} + S) \times 2^S \end{aligned}$$

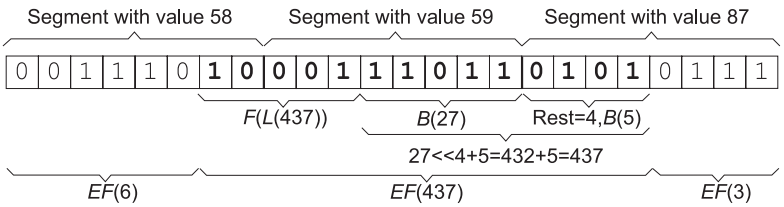


Figure 11: An example of a partially decoded number; this number must be shifted

Example 17. *Let us consider an example in Figure 11. We access the following records of the mapping table during the decoding:*

$$MAP[Segment = 58, State = 0] = \\ \{NewState = 11, Rest = 0, U = 0, OutputCount = 1, Numbers = \{6\}\}$$
$$MAP[Segment = 59, State = 11] = \\ \{NewState = 4, Rest = 4, U = 27, OutputCount = 0, Numbers = \{\}\}$$
$$MAP[Segment = 87, State = 4] = \\ \{NewState = 0, Rest = 0, U = 0, OutputCount = 2, Numbers = \{5, 3\}\}$$

After reading the first segment we directly obtain the number 6 *in the mapping table. It is immediately stored in the result array. The automaton is currently in the state 11. Obviously, we have two 0-bits of* $F(L(n))$ *in the first segment. Now, the second segment with the value 59 is read. In the mapping table, we see that it represents the incomplete number* $U = 27$ *and there are 4 bits (see the Rest value) in the next segment to complete the encoded value. The Fibonacci shift is applied and the result is stored in* n*:* $n = 27 >> 4 = 432$*. The next segment starts with the remaining 4 bits; the sequence* 0101 *is read. In this state, we get the number* 5 *which is added to* n*; we compute* $n = 432 + 5 = 437$*. This number is stored in the result array together with the next number* 3 *stored in* $Numbers$*.*

7. Experimental Results

7.1. Introduction

In our experiments, we tested all proposed fast decoding algorithms and compared these algorithms with conventional bit-oriented algorithms. The performance has been tested for various data collections. The experiments were executed on a PC with AMD Opteron 1.8GHz, 32 GB RAM; Windows Server 2008, 64 bit.

We used data collections with the maximum value $2^{64}-1$, it means the maximum value for the 64 bit-length number. All fast decoding algorithms utilize the 8 bit-length segment size, in other words $S = 8$. Table 9 summarizes the number

Table 9: The number of automaton states for 32 and 64 bit-length numbers and 8 bit-length segment size

Encoding type		Elias-delta	Fibonacci $m = 2$	Fibonacci $m = 3$	Elias-Fibonacci
The number of states	32-bit	45	2	3	64
	64-bit	78	2	3	151
Mapping table length	32-bit	11,520	512	768	16,348
	64-bit	19,968	512	768	38,656

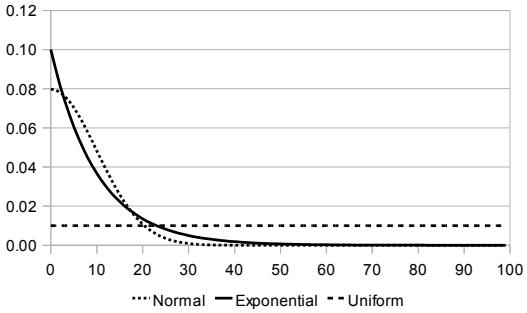


Figure 12: The uniform distribution, normal distribution for $\sigma^2 = 100$, and exponential distribution for $\lambda^2 = 100$

of states and the mapping table length for all fast algorithms. We used both random and real data collections. In the case of random data collections, uniform, normal, and exponential distribution functions are used (see Figure 12). Each random collection includes 10 million numbers. The test collections were as follows:

- Uniformly distributed random collections:
  - 8 bit-length numbers in the range of $\langle 0; 2^8 - 1\rangle$
  - 16 bit-length numbers in the range of $\langle 2^8; 2^{16} - 1\rangle$
  - 32 bit-length numbers in the range of $\langle 2^{16}; 2^{32} - 1\rangle$
  - 64 bit-length numbers in the range of $\langle 2^{32}; 2^{64} - 1\rangle$
- Normally distributed random 32 bit-length numbers – a collection of random numbers in the range of $\langle 0; 2^{32} - 1\rangle$; we have used normal distribution function $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}}$, where $\sigma^2 = 2^{32}$.
- Exponentially distributed random 32 bit-length numbers – a collection of random numbers in the range of $\langle 0; 2^{32} - 1\rangle$; we have used exponential distribution function $f(x) = \frac{1}{\lambda} e^{\frac{-x}{\lambda}}$, where $\lambda^2 = 2^{32}$.

- Real collection – the Bible (King James version) in English included in the Canterbury corpus [21]. We extracted all terms regardless of lower/upper case letters. In this way, we identified 766,131 terms. The collection of terms remained unsorted; the order of terms remained the same as in Bible. After that we created the dictionary of all unique terms with the empirical probability of occurrence. The length of the dictionary is 13,744 terms. The smallest codes were assigned with the highest probability of term occurrence.

7.2. Results for Uniform Random Collections

In the first test, we measured the decoding time as well as the coded data size for uniform random collections. Each test was executed several times with different random seed numbers and we used the average values. The results of decoding time are summarized in Table 10 and Figures 13(a) and (b). The coded size and average number of bits per coded number are shown in Table 11 and Figure 13(c).

Table 10: Decoding time results for uniform random collections [s]

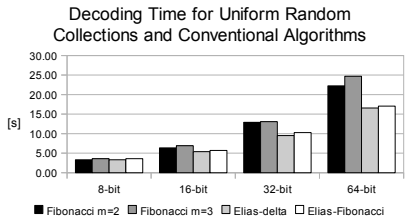
Data collection	**Conventional decoding algorithms**			
	Fibonacci $m = 2$	**Fibonacci** $m = 3$	**Elias-delta**	**Elias-Fibonacci**
8-bit	3.32	3.59	3.31	3.60
16-bit	6.35	6.90	5.41	5.69
32-bit	12.90	13.06	9.52	10.26
64-bit	22.23	24.70	16.57	17.07
Avg.	11.20	12.06	**8.70**	9.16

Data collection	**Fast decoding algorithms**			
	Fibonacci $m = 2$	**Fibonacci** $m = 3$	**Elias-delta**	**Elias-Fibonacci**
8-bit	0.80	0.67	0.60	0.57
16-bit	1.42	1.11	0.90	0.85
32-bit	2.74	2.02	1.49	1.35
64-bit	5.33	4.67	2.61	2.51
Avg.	2.57	2.12	1.40	**1.32**

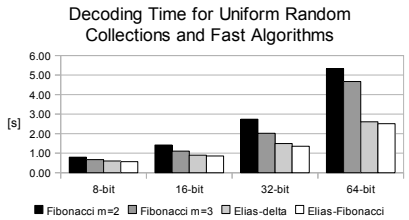
Although the most efficient compression ratio was achieved by the Fast Fibonacci code of order 3, the fast decoding algorithm of the newly proposed Elias-Fibonacci code outperforms all other fast algorithms from the decoding time point of view. Fast Fibonacci decoding algorithms of both orders utilize the Fibonacci shift operation, which is more time consuming compared to a common shift operation applied in the case of Elias-delta as well as Elias-Fibonacci

Table 11: Coded size and average number of bits per coded number for uniform random collections

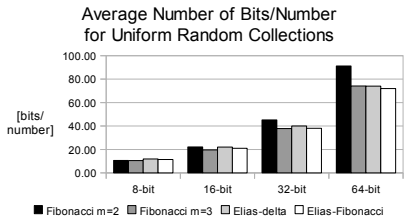
Data	**Fibonacci** $m = 2$		**Fibonacci** $m = 3$		**Elias-delta**		**Elias-Fibonacci**	
	[MB]	[b./n.]	[MB]	[b./n.]	[MB]	[b./n.]	[MB]	[b./n.]
8-bit	12.7	10.6	**12.5**	**10.5**	14.2	11.9	13.6	11.4
16-bit	26.5	22.2	**23.4**	**19.6**	26.3	22.0	25.0	21.0
32-bit	53.9	45.2	**45.0**	**37.8**	47.7	40.0	45.3	38.0
64-bit	108.8	91.3	88.4	74.2	88.2	74.0	**85.8**	**72.0**
Avg.	50.5	42.3	**42.3**	**35.5**	44.1	37.0	42.4	35.6



(a)



(b)



(c)

Figure 13: Results for the uniform random data collections: (a) Decoding time for conventional algorithms (b) Decoding time for fast algorithms (c) The average number of bits per number

decoding algorithms. Consequently, Fibonacci of order 3 may be used in cases where we prefer a higher compression ratio rather than a lower decoding time. On the other hand, Elias-Fibonacci may be used in cases where we prefer a lower decoding time rather than a higher compression ratio. Let us suppose the average results for all uniform random collections. The improvement of the compressed data size for the Fibonacci of order 3 code is approximately 4% compared to the Elias-Fibonacci code. On the other hand, decoding time of the Elias-Fibonacci code is reduced by 32% compared to the Fibonacci of order 3 code.

In the case of the 8 bit-length data collection, fast algorithms do not utilize shift operations; therefore, the decoding times are very similar. In this case,

Table 12: Acceleration ratio between conventional and fast decoding algorithms for uniform random data collections

Data collection	**Acceleration ratio**			
	Fibonacci $m = 2$	Fibonacci $m = 3$	Elias-delta	Elias-Fibonacci
8-bit	4.18	5.32	5.48	6.33
16-bit	4.48	6.23	6.04	6.67
32-bit	4.71	6.47	6.37	7.59
64-bit	4.17	5.29	6.36	6.80
Avg.	4.39	5.83	6.06	**6.85**

Fibonacci codes of any order are the most efficient choice because these codes provide higher compression ratios than Elias-delta or Elias-Fibonacci codes.

In Table 12, we see the acceleration ratio between conventional and fast decoding algorithms for uniform random data collections. The proposed fast algorithms are 5× more efficient on average than conventional bit-oriented algorithms.

7.3. Results for Normal and Exponential Random 32 Bit-length Collections

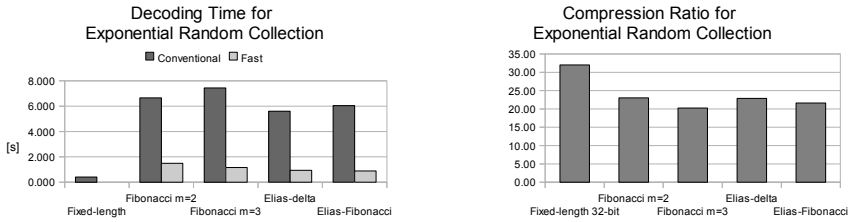
Further, we tested normal and exponential random 32 bit-length data collections. The results are depicted in Tables 13 and 14 and Figures 14 and 15. Compression ratios are measured by the average number of bits per coded number. We see that these results support the conclusions of the previous test; the fast algorithms are approximately 5× faster than conventional algorithms. The Fibonacci of order 3 code is the most efficient coding from the compression ratio point of view, and the Fast Elias-Fibonacci decoding algorithm is the most efficient from the time decoding point of view.

Table 13: Encoded size, compression ratio, and decoding time for the exponential random collection

Code	**Encoded Size** [MB]	**Compression Ratio** [bits/number]	**Decoding Time** [s]	
			Conv.	**Fast**
Fixed-length 32-bit	38.15	32.00	**0.41**	-
Fibonacci $m = 2$	27.44	23.02	6.66	1.49
Fibonacci $m = 3$	**24.14**	**20.25**	7.44	1.16
Elias-delta	27.27	22.87	5.60	0.94
Elias-Fibonacci	25.75	21.60	6.04	**0.89**

In this test, we also compared fast algorithms with the fixed-length memory reading of 32 bit-length integers. We measured the time required for copying data from one memory into another memory. The decoding time of the Fast

Elias Fibonacci decoding algorithm is 2.2× less efficient than the fixed-length memory reading. However, the encoded size for the Fibonacci of order 3 code is 2.5× lower compared to the fixed memory.

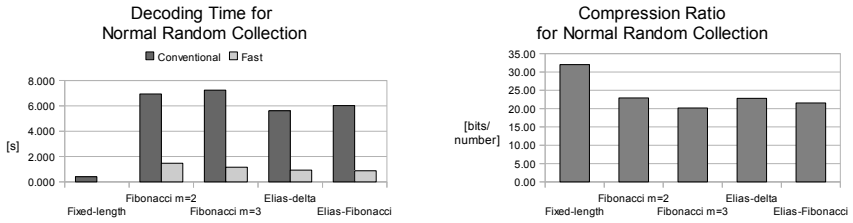


(a) (b)

Figure 14: Decoding time (a) and compression ratio (b) for the exponential random collection

Table 14: Encoded size, compression ratio, and decoding time for normal random collection

Code	**Encoded Size** [kB]	**Compression Ratio** [bits/number]	**Decoding Time** [s]	
			Conv.	**Fast**
Fixed-length 32-bit	38.15	32.00	**0.41**	-
Fibonacci $m = 2$	27.29	22.90	6.94	1.47
Fibonacci $m = 3$	**24.03**	**20.16**	7.24	1.16
Elias-delta	27.19	22.81	5.62	0.93
Elias-Fibonacci	25.66	21.53	6.02	**0.88**



(a) (b)

Figure 15: Decoding time (a) and compression ratio (b) for the normal random collection

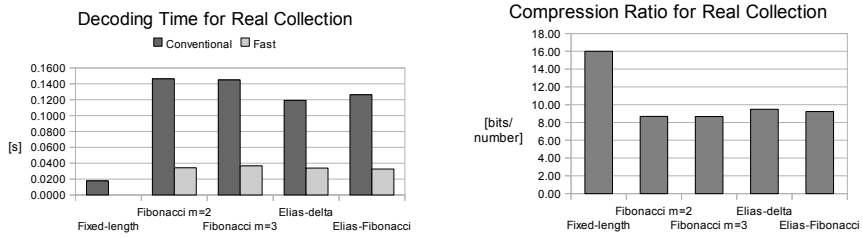
7.4. Results for Real Collection

Although we have shown the time decoding efficiency of the introduced fast decoding algorithms, efficiency from the compression ratio point of view has not

yet been presented. Therefore, we used the proposed real data collection in our last test. We compared the decoding times and compression ratios with fixed-length 16-bit integers. The results are summarized in Table 15 and Figure 16. The most efficient compression ratio was achieved by the Fibonacci of order 3 code. The improvement of the compression ratio is 60%. Once again, fast algorithms are approximately 5× faster than conventional algorithms. Obviously, the decoding time for the fixed-length code is more eficient than the decoding time of other codes. The most efficient code, from the decoding time point of view, was Fast Elias-Fibonacci. In this case, the decoding time is approximately 50% slower than reading fixed-length 16-bit integers.

Table 15: Encoded size, compression ratio, and decoding time for the real data collection

Code	**Encoded Size** [kB]	**Compression Ratio** [bits/term]	**Decoding Time** [s]	
			Conv.	**Fast**
Fixed length 16-bit	1,502	16	**0.0178**	–
Fibonacci $m = 2$	892	8.69	0.1462	0.0343
Fibonacci $m = 3$	**890**	**8.66**	0.1449	0.0367
Elias-delta	974	9.48	0.1192	0.0339
Elias-Fibonacci	948	9.23	0.1262	**0.0328**



(a) (b)

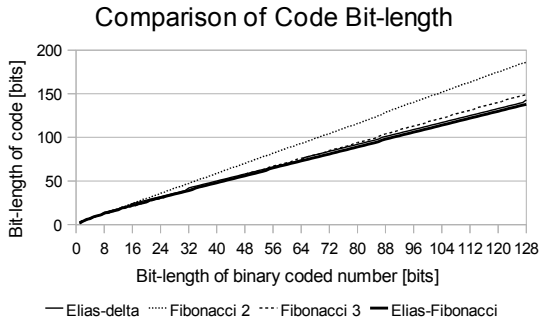
Figure 16: Decoding time (a) and compression ratio (b) for the real collection

The Fibonacci code of order 3 provides a less efficient decoding time than the Fibonacci code of order 2 in all tests, because this algorithm uses the final adjustment of the value which escalates the decoding time.

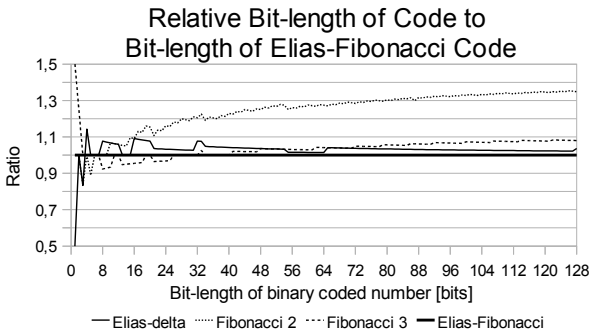
7.5. Comparison of Code Bit-length

Evidently, the Elias-Fibonacci code provides a more efficient compression ratio for large numbers than other codes; see results for the 64 bit-length data collection in Table 11 as an example. This property can be seen more clearly in a comparison of bit-lengths of all codes for various number domains in Figure 17.

All codes of Elias-Fibonacci are shorter than other compared codes for numbers longer than 26 bits.



(a)



(b)

Figure 17: Comparison of the length of the code (a) and the relative length of the code to the length of the Elias-Fibonacci code (b) for various lengths of binary coded numbers

8. Conclusion

In this article, we introduced fast decoding algorithms for Elias-delta, Fibonacci of orders 2 and 3, and Elias-Fibonacci codes and we introduced a theoretical background of these fast algorithms. Whereas Elias-delta and Fibonacci of orders 2 and 3 are well-known codes, the Elias-Fibonacci code has been introduced in this paper. Fast algorithms are based on the finite automaton and precomputed mapping table. Since the number of states is rather high, we introduced two types of automaton reduction: grouping similar states and shift operations. The Fast Fibonacci decoding algorithm utilizes the newly proposed Fibonacci shift operation, whereas Elias-delta and Elias-Fibonacci utilize a common shift operation. In our experiments, all fast decoding algorithms

are 5× faster on average than the conventional algorithms. The highest compression ratio for lower numbers was achieved by the Fast Fibonacci code of order 3; however, the newly proposed Elias-Fibonacci code achieved the most efficient decoding time for all tested data collections. Codes of Elias-Fibonacci are shorter than other compared codes for numbers longer than 26 bits. In future work, we plan to develop fast coding algorithms for all presented codes.

Acknowledgment

This work is supported by grants of GACR No. P202/10/0573 and 201/09/0990 and the Ministry of Education MSM 6198910007, Czech Republic.

The Bibliography

[1] T. Acharya, J.F. Jájá, An on-line variable-length binary encoding of text, Information Sciences 94 (1996) 1–22.

[2] A. Apostolico, A. Fraenkel, Robust Transmission of Unbounded Strings Using Fibonacci Representations, IEEE Transactions on Information Theory 33 (1987) 238–245.

[3] R. Bayer, E. McCreight, Organization and Maintenance of Large Ordered Indexes, Acta Informatica 1 (1972) 173–189.

[4] P. Elias, Universal Codeword Sets and Representations of the Integers, IEEE Transactions on Information Theory IT-21 (1975) 194–203.

[5] P. Ferragina, G. Manzini, An experimental study of a compressed index, Information Science 135 (2001) 13–28.

[6] A. Fraenkel, S. Klein, Robust Universal Complete Codes As Alternatives to Huffman Codes, Technical Report Tech. Report CS85-16, Dept. of Appl. Math., The Weizmann Institute of Science, Rehovot, 1985.

[7] J. Goldstein, R. Ramakrishnan, U. Shaft, Compressing Relations and Indexes, in: Proceedings of the 14th International Conference on Data Engineering, ICDE 1998, IEEE Computer Society, Los Alamitos, CA, USA, 1998, p. 370.

[8] S.W. Golomb, Run-Length Encodings, IEEE Transactions on Information Theory IT-12 (1966) 399–401.

[9] A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Annual Meeting, Boston, USA, ACM Press, June 1984, pp. 47–57.

[10] D. Huffman, A Method for the Construction of Minimum Redundancy Codes, Proceedings of the IRE 40 (1952) 1098–1101.

[11] S.T. Klein, Fast Decoding of Fibonacci Encoded Texts, in: Proceedings of the International Data Compression Conference, DCC 2007, IEEE Computer Society, USA, 2007, p. 388.

[12] S.T. Klein, M.K. Ben-Nissan, Using Fibonacci Compression Codes as Alternatives to Dense Codes, in: Proceedings of the International Data Compression Conference, DCC 2008, IEEE Computer Society, USA, 2008, pp. 472–481.

[13] S.T. Klein, M.K. Ben-Nissan, On the Usefulness of Fibonacci Compression Codes, The Computer Journal 53 (2010).

[14] Leonardo of Pisa (known as Fibonacci), Liber Abaci, 1202.

[15] U. Manber, A text compression scheme that allows fast searching directly in the compressed file, ACM Transaction of Information Systems 15 (1997) 124–136.

[16] F. Marcelloni, M. Vecchio, Enabling energy-efficient and lossy-aware data compression in wireless sensor networks by multi-objective evolutionary optimization, Information Sciences 180 (2010) 1924 – 1941. Special Issue on Intelligent Distributed Information Systems.

[17] A. Moffat, A. Turpin, Compression and coding algorithms, Springer, 2002.

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

[19] G.H. Ong, S.Y. Huang, A data compression scheme for Chinese text files using Huffman coding and a two-level dictionary, Information Sciences 84 (1995) 85–99.

[20] H. Plantinga, An Asymmetric, Semi-adaptive Text Compression Algorithm, in: Proceedings of the International Data Compression Conference, DCC 1994, IEEE Computer Society, 1994.

[21] M. Powell, The Canterbury corpus, 2001. `http://www.corpus.canterbury.ac.nz/` [Online; accessed 10-October-2010].

[22] J.H. Reif, J.A. Storer, Optimal encoding of non-stationary sources, Information Sciences 135 (2001) 87–105.

[23] L. Rueda, B.J. Oommen, A fast and efficient nearly-optimal adaptive Fano coding scheme, Information Sciences 176 (2006) 1656–1683.

[24] D. Salomon, Data Compression: The Complete Reference, Third Edition, Springer–Verlag, New York, 2004.

[25] D. Salomon, Variable-length Codes for Data Compression, Springer-Verlag, 2007.

[26] H. Samet, Data Structures for Quadtree Approximation and Compression, Communications of the ACM archive 28 (September 1985) 973–993.

[27] P. Skibinski, PPM with the extended alphabet, Information Sciences 176 (2006) 861–874.

[28] J. Walder, M. Krátký, R. Bača, Benchmarking Coding Algorithms for the R-tree Compression, in: Proceedings of the Dateso 2009 Annual International Workshop on Databases, Texts, Specifications and Objects, volume 471 of *CEUR-WS.org - CEUR Workshop Proceedings*, pp. 32–43.

[29] Wikipedia, Wikipedia, the free encyklopedia, 2010. `http://en.wikipedia.org/` [Online; accessed 10-October-2010].

[30] H. Williams, J. Zobel, Compressing Integers for Fast File Access, The Computer Journal 42 (1999) 193–201.

[31] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes, Compressing and Indexing Documents and Images, 2nd edition, Morgan Kaufmann, 1999.

[32] H. Zhao, C. Kit, Integrating unsupervised and supervised word segmentation: The role of goodness measures, Information Sciences 181 (2011) 163 – 183.