Reverse Multi-Delimiter Codes in English and Ukrainian Natural Language Text Compression

Igor Zavadskyi[a] and Viktoriia Zavadska[b]

[a] *Taras Shevchenko National University of Kyiv, 4d Glushkova ave, 03022, Kyiv, Ukraine*
[b] *National Technical University of Ukraine 'Igor Sikorsky Kyiv Polytechnic Institute', 37 Prosp. Peremohy, 03056, Kyiv, Ukraine*

Abstract

A new "reverse" version of multi-delimiter data compression codes is presented. A fast decoding algorithm for these codes is discussed. The experiments provided to compare the compression ratio and decoding time of different codes applied to Ukrainian and English natural language texts. The optimal parameters of reverse multi-delimiter codes for Ukrainian natural language text compression are identified.

Keywords [1]

Natural language, Ukrainian, Compression, Code, Multi-Delimiter

1. Introduction

Natural language text compression is one of the key components of modern information retrieval systems. From a perspective of an input alphabet structure, all compression methods in this field can be divided into two groups. The first consists of methods operating individual characters as alphabet elements, while methods of the second group rely on words of a text as atomic units. Generally, the latter methods provide a better compression ratio, and we focus our attention on them. This approach implies using a *dictionary*, i.e., some mapping between the set of words of a text and the set of codewords. In terms of compression ratio, it is not important how this mapping is implemented technically; the only requirement is that shorter codewords correspond to more frequent words of a text. Several known codes provide the compression ratio close to the theoretical limit defined by Shannon's entropy. This refers to arithmetic encoding, codes based on asymmetric numeral systems [4], and, to some extent, Huffman codes [5].

However, apart from the compression ratio, a number of other requirements should be taken into account, for which the above-mentioned codes are not well-suited.

- Encoding/decoding speed. The decoding is more important since it is often performed "online", many times for a text encoded "offline" only once.
- Synchronizability. This means that possible error in the codeword sequence has a limited area of propagation.
- Compressed search. Possibility of searching information in the compressed file without it decompression can significantly improve the performance of information retrieval systems.

In order to provide fast decoding, a data structure with low access time should be used to store the dictionary. For these purposes, the most efficient data structure is the array with integer indices. Then, at each iteration of the decoding loop, we need to extract a codeword from a coded bitstream, convert this codeword to a number, and output the corresponding dictionary element. The mentioned extraction and conversion can be done in parallel, and thus the decoding algorithm, in fact, consists in mapping a bitstream of codewords to a sequence of integers. Since a bitstream is divided into bytes in computer memory, and the processor operates with bytes or even machine words, it is reasonable to explain a decoding algorithm in terms of byte-level or machine-word-level operations.

Information Technology and Implementation (IT&I-2021), December 01–03, 2021, Kyiv, Ukraine
EMAIL: ihorzavadskyi@knu.ua (A. 1); ptits@ukr.net (A. 2)
ORCID: 0000-0002-4826-5265 (A. 1); 0000-0002-6411-4926 (A. 2)

There are two ways to achieve this goal. The first consists of constructing codewords from the whole number of bytes only. Such codes are called byte-aligned. The most advanced representatives of this family are (s,c)-dense codes (SCDC) [2] and codes with restricted prefix properties (RPBC) [3]. They make possible ultra-fast decoding, however, at the cost of compression ratio, which becomes 14–18% beyond the entropy when English texts are compressed. The other way is to use lookup tables, which allows us to decode whole bytes or other blocks of bits at one iteration of a decoding loop. This approach is implemented for Fibonacci codes in [6] and recently invented multi-delimiter / reverse multi-delimiter (MD) codes in [1]. These codes provide a better compression ratio, 2–5% (multi-delimiter codes) or 4–7% (Fibonacci codes) beyond the entropy. And although the proposed fast methods of their decoding are 25–100% slower than those ones for byte-aligned codes, in general, MD-codes are at attractive point in the tradeoff between compression ratio, decoding speed, and other useful code properties.

Nonetheless, for MD-codes, one important problem was not clarified. It weights toward the efficient dictionary indexing. Namely, the encoding mapping ψ, described in [1], has one quite essential disadvantage: the codewords $\psi_1, \psi_2, \ldots$ are not sorted in ascending order of their lengths. It follows that the codewords $\psi_1,\ldots,\psi_n$ do not constitute the set of n shortest codewords. However, in order to gain the maximum compression ratio, one should use the n shortest codewords $\psi_{i_1},\ldots, \psi_{i_n}$ where i_n could be much greater than n. Thus, an array having non-empty elements with indices $i_1,\ldots,i_n$ could be very sparse and impractical even for coding relatively short texts containing 2000-3000 different words. Although some workarounds to resolve this issue were presented in [7], the best option is to construct a monotonic encoding mapping ψ (i.e., if $i_1>i_2$ then $\psi_{i_1} \geq \psi_{i_2}$). However, for MD-codes, it seems to be problematic to build the inverse mapping based on processing bits of a codeword from left to right, since processing a part of a codeword doesn't give much information about the position of a whole codeword in the dictionary. One can assume that this mapping can be constructed for a non-prefix code, if shorter codewords are prefixes of longer ones. Let us discuss uv — the concatenation of a codeword u with some suffix v. To decode it, we can proceed as follows:

1. in the ordered set of all codewords obtain the index of u;
2. increase this index by some value which depends on v and can be calculated by the special algorithm.

We call this approach "decoding in parts". In particular, it allows us to process codewords that occupy several bytes in a byte-by-byte manner.

Note that if we write bits of codewords of a multi-delimiter code in the reverse order, then we get a non-prefix but a uniquely decodable code. Indeed, an encoded file can be decoded from right to left as a multi-delimiter code. Also, it can be uniquely decoded from left to right, since in this code all delimiters remain the same as in the source multi-delimiter code, however, they are placed in the beginnings of codewords instead of their ends. Such "reverse" multi-delimiter codes are the main subject of this presentation. For modified this way multi-delimiter codes, the most remarkable property we discover is the existence of a monotonic encoding mapping from the set of natural numbers to the set of codewords, for which a simple inverse decoding mapping based on processing bits from left to right can be easily constructed.

In this paper, we give the definition of the reverse multi-delimiter codes (Section 2) and describe a fast method of their decoding implementing the "decoding in parts" principle (Section 3). In Section 4 we discuss the results of experiments measuring the compression ratio and decoding speed of different reverse multi-delimiter, Fibonacci, and byte-aligned codes applied to compression of English and Ukrainian texts. Also, we discuss the specificities and differences between English and Ukrainian natural language text compression. In Section 5 we make some final conclusions.

2. Codeword Set

Let $M = \{m_1, \ldots, m_t\}$ be a set of integers, given in the ascending order, $0 < m_1 <\ldots< m_t$. The reverse multi-delimiter (RMD) code $R_{m_1, \ldots, m_t}$ 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:

3. a word starts with the prefix $01^{m_i}0$ for some $m_i \in M$;
4. for any $m_i \in M$ a word does not end with the sequence 01^{m_i};

5. for any $m_i \in M$ a word can contain the sequence $01^{m_i}0$ only as a prefix.

This implies that delimiters in $R_{m_1, \ldots, m_t}$ are the sequences of the form $01^{m_i}0$. However, the code also contains shorter words of the form 01^{m_i} that constitute delimiters when appended by the leading zero of a next codeword.

Now let us construct a monotonous encoding mapping that maps the set of natural numbers to the set of codewords of the RMD-code $R_{m_1, \ldots, m_t}$. Let $K_{k_1, \ldots, k_q}$ be the sequence of integers in the range [0; m_t+1] given in the ascending order that don't belong to M. For instance, K={0,1,3,6} for the code $R_{2,4,5}$. Note that each codeword of $R_{m_1, \ldots, m_t}$ has the following structure: it consists of a prefix 01^{m_i}, for some $m_i \in M$, which can be followed by some groups of bits having the form 01^s, where $s \in K$ or $s>k_q$. The inverse statement is also correct: a bit sequence having the described structure constitutes a codeword.

Therefore, in the code $R_{m_1, \ldots, m_t}$ any codeword of the length L can be constructed in one (and only one) of the following ways:

1. it is a word of the form 01^{m_i}, $i = 1, \ldots, t$;
2. it is composed of a codeword of the length L-s-1 followed by the sequence 01^s, $s \in K$;
3. it is composed of a codeword of the length L-1 with a suffix 01^r, $r > m_t$ appended by a single '1' bit.

Following these facts, we formulate a principle of constructing the ordered set of codewords of the length L assuming the shorter codeword sets have been already built.

1. For i from 1 to q, replicate the sets of codewords of lengths L-k_i-1 and append the sequences of the form 01^{k_i}, $k_i \in K$, to them.
2. Replicate the codewords of the length L-1 with a suffix 01^r, $r>m_t$, and append a single '1' bit to all elements of this set.
3. If $L=m_i+1$, $i \in \{1, \ldots, t\}$, append the word 01^{m_i} to the codeword set.

If we apply this approach gradually to form the sets of codewords of lengths m_1+1, m_1+2,…, we obtain the whole set of codewords ordered by their lengths. For example, the set of 8-bit or shorter codewords belonging to the code $R_{2,4,5}$ is given in Fig. 1. Also, it demonstrates how the words of the length 8 are constructed from the words of lengths 7, 6 and 4 by appending bit sequences 0, 01, and 0111 respectively (rule 1). Three codewords highlighted in grey are constructed by applying the rule 3. The shortest codeword that is constructed by applying the rule 2 has the length 11; its construction is shown in the left bottom part of the figure. Let us note that the reverse multi-delimiter codes possess all properties of MD-codes, e.g. unique decodability, completeness, and universality as well as their asymptotic densities, because $R_{m_1, \ldots, m_t}$ contains the same number of codewords of a given length as the "direct" multi-delimiter code $D_{m_1, \ldots, m_t}$. For the original version of multi-delimiter codes these properties were proven in [1]. The completeness means that no codeword can be added to codeword set so that the code remains uniquely decodable. And a code is universal in the sense of Elias [8] if the average codeword length is no more than constant times longer than the average codeword length of the optimal code for any source alphabet characters distribution.

The number of short codewords of some reverse multi-delimiter codes and, for comparison, Fibonacci code Fib3 are given in Table 1 (in natural language text compression the code Fib3 is considered to be the most efficient one among Fibonacci codes). As seen, the RMD-codes can contain rather bigger number of short codewords than Fibonacci code Fib3. Of course, this superiority is compensated by the number of long codewords. However, the number of short codewords is rather more important when it comes to compression of real world textual databases.

3. Encoding and Decoding

Once the set of codewords c_1, c_2, … ordered by their lengths is built, the encoding of a text becomes trivial. It only requires ordering words of a text to be encoded according to descending order of their frequencies. The word w_i in this ordered set gets the codeword c_i.

The decoding is rather more sophisticated. In fact, it concludes recognizing a codeword c_i in the flow of codewords and calculating its index i in the set of codewords built by the principle given in the previous section. Note that the code $R_{m_1, \ldots, m_t}$ is a regular language in the binary alphabet. Thus, it

can be recognized by a finite automaton, which processes bits of a codeword from left to right. We can apply this automaton to calculate the index of a codeword in the ordered set of codewords.

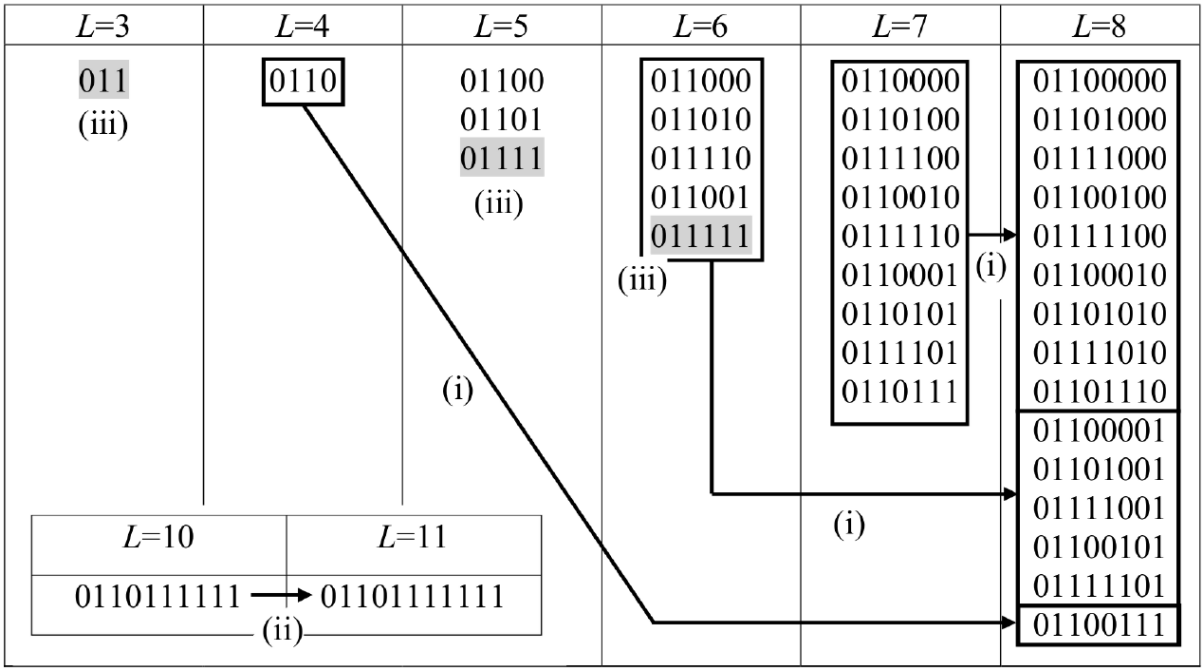

Figure 1: Construction of the codeword set of the code $R_{2,4,5}$

Table 1

Number of codewords of length ≤*n* in codes with delimiters

Code\n	3	4	5	6	8	10	15	20
Fib3	1	2	4	8	28	96	2031	42762
$R_{2,-\infty}$	1	3	7	14	46	133	1581	17690
$R_{3,-\infty}$	0	1	3	7	30	110	2413	50941
$R_{2,4,-\infty}$	1	2	5	10	37	122	2113	35283

In the sequel, we concentrate on the "infinity" versions of RMD-codes, e.g. $R_{2,-\infty}$ or $R_{2,4,-\infty}$, as they demonstrate the best compression ratio. The "infinity" $R_{2,-\infty}$ code is built on all delimiters containing 2,3,… ones. However, in practice it is enough to limit the length of delimiters by some relatively big number, for example 21, since the difference in compression ratio between $R_{2,21}$ and $R_{2,22}$ is negligibly small.

The decoding automaton for the code $R_{2,-\infty}$ is shown in Fig. 2. It processes codewords in the flow starting in the state marked by the filled circle and finishing in the state 1 after processing a delimiter in the beginning of some codeword. Thus, for the decoding to be successfully completed, an encoded file must be appended with some delimiter, e.g. 01110. The automaton recognizes codewords bit-by-bit and calculates the values of two variables: L - a length of a codeword and p – a resultant index of a codeword. On each transition, an input bit is indicated before the vertical bar and operations on L and p are indicated after it; the order of operations is important.

Let us explain how the automaton works. Any codeword of $R_{2,-\infty}$ can be represented as a concatenation of bit sequences of the form 01^t, where $t \geq 0$. After processing any such sequence and '0' after it the automaton comes to the state 0. If $t \geq 2$, the bit sequence 01^t can be only the prefix of a codeword. Thus, after processing thew last 1 in the sequence 011, the automaton comes to the state 2 from the state 1, outputs the resultant index p of the previous codeword and initializes L with a new value 3=|011|. On all other transitions, except for the initial one, L is incremented by 1.

When the automaton comes from the state 2 to the state 0, this means that the prefix of a codeword of a form $01^{L-1}0$ is processed and p should be initialized with the index of the codeword 01^{L-1} in the codeword set, counted from 0. As seen in Fig. 1, this index is equal to n_L-1, where n_L is the number of codewords of length not greater than L.

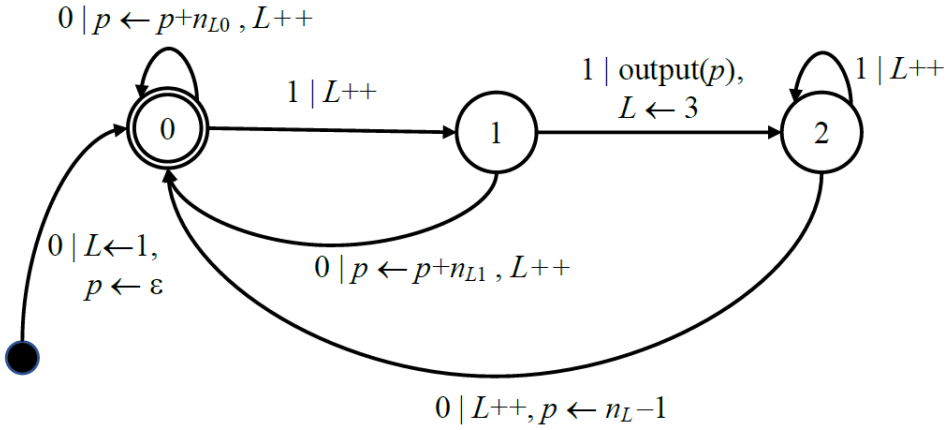

Figure 2: The decoding automaton for $R_{2-\infty}$

After processing the bit sequences 00 or 010, the automaton makes transitions to the state 0 from the states 0 and 1 respectively and adds to the codeword position p the values n_{L0} or n_{L1} respectively. n_{Li} is equal to the shift of a codeword in the list of all codewords due to appending the bit sequence of the form 01^i, $0 \le i \le 1$, when the length of a resultant codeword is L. Note that the aforementioned principle of constructing the codeword list guarantees that appending the same group 01…1 to any codeword of a given length shifts this codeword to the same distance. The values n_{Li} and n_L can be calculated in the preprocessing stage together with the ordered codeword set.

For different tested texts (see Section 4), codes $R_{3-\infty}$ and $R_{2,4-\infty}$ also provide the best compression ratio, apart from $R_{2-\infty}$. Their decoding automatons are shown in Fig. 3 and Fig. 4 respectively. The automaton for $R_{3-\infty}$ is pretty like that one for $R_{2-\infty}$, although having the extra state 3. The automaton for $R_{2,4-\infty}$ also recognizes the sequence 0110 as the beginning of a codeword. In this case it makes the transition from the state 2 to the state 0, outputs the result p of decoding of the previous codeword, assigns 4 to L, since 4 bits of a new codeword are processed, and 0 to p, because the codeword 011 occupies position 0 in the ordered codeword set.

Applying the decoding automaton to processing an encoded text bit-by-bit could be quite slow. However, we can make the byte "quantification" of an automaton. It is done by iterative reading a fixed integral number of bytes and producing the corresponding output numbers.

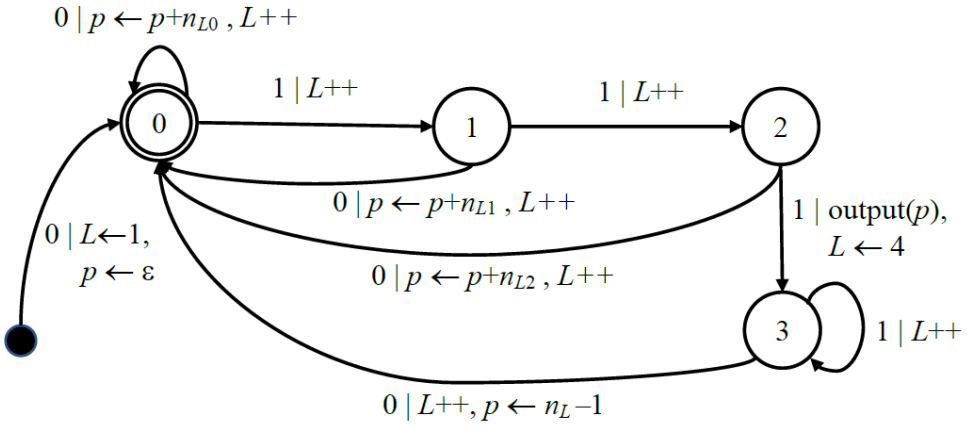

Figure 3: The decoding automaton for $R_{3-\infty}$

The principle of quantification is the same as described in [1]. The quantified automaton is given as a lookup table TAB[a_{start}][L_{start}][x] consisting of tuples (p, a_{fin}, L_{fin}), where x is a portion of bytes of the encoded file, a_{start} is the state of the automaton before processing x, a_{fin} is the state it comes to after processing x, L_{start} is the length of an already decoded part of the last codeword, which is under processing when the decoding of x starts, L_{fin} is the length of a decoded part of the last codeword generated from x and p is the output generated by the automaton while processing x.

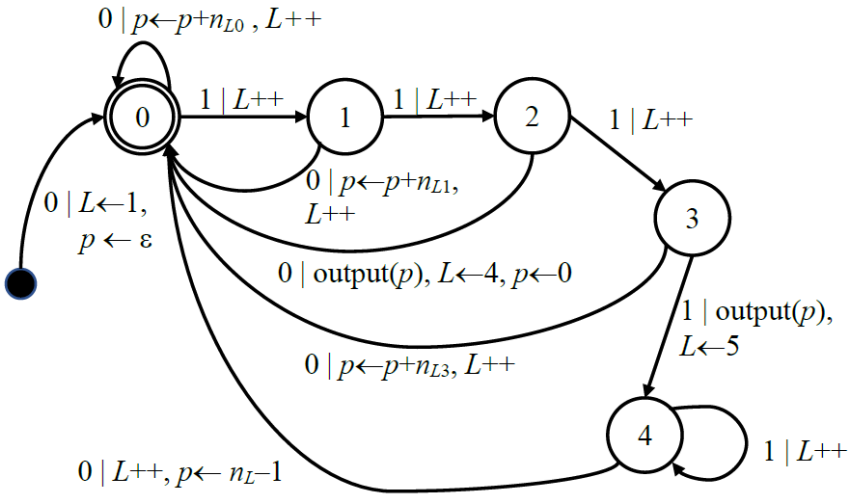

Figure 4: The decoding automaton for $R_{2,4,-\infty}$

The byte-aligned decoding algorithm for any code with the shortest delimiter 0110 is given below. Its lookup table is built by the automatons from Fig. 2 – Fig. 4, quantified to read bytes of the encoded text. In order to fast decoding, we use the series of one-dimensional lookup tables $TAB_j[x]$ instead of one three-dimensional. The number j of the lookup table depends on the parameters a_{start} and L_{start}, e.g. it can be obtained as $5L_{start} + a_{start}$ (since 5 is the number of automaton states). Also, the tuples in the lookup table contain only the number j_{fin} instead of a pair (a_{fin}, L_{fin}). The output is represented by numbers p_1, p_2, p_3, p_4 – the indices of decoded codewords or their decoded parts in the array of all codewords. We have 4 values here, since no more than 4 codewords can be decoded, fully or partially, while processing one byte of a code if its shortest delimiter is 0110. Indeed, the only case, when the parts of 4 codewords are included into 8-bit sequence, is x 011 011 0. This byte includes two shortest codewords 011, prepended with the ending bit x of some codeword and appended by the first bit 0 of another codeword. Also, note that the value p_1 can also represent the shift of the codeword uv in the list of codewords towards its already decoded prefix u. Let us note that this algorithm is suitable for decoding any reverse multi-delimiter code with the shortest delimiter containing two ones: 0110.

***Input*:** *Encoded text*
***Output*:** *The indices of decoded words in the array of codewords*
$i \leftarrow 0$;
$j \leftarrow 0$;
$p \leftarrow 0$;
while(*i*< *length of encoded text*)
    (p_1, p_2, p_3, p_4, j) ← $TAB_j[Text[i]]$;
    $p \leftarrow p+p_1$;
    if($p_2 \geq 0$)
        output(p);
        if($p_3 \geq 0$)
            output(p_2);
            if($p_4 \geq 0$)
                output(p_3);
                $p \leftarrow p_4$;
            else
                $p \leftarrow p_3$;
        else
            $p \leftarrow p_2$;
    $i \leftarrow i+1$;

Algorithm 1: The byte-aligned decoding algorithm for $R_{2,x}$

It is easy to calculate the size of the lookup table of the presented byte aligned decoding algorithm. For the larger English text, in our experiments containing 288K different words, the maximum codeword length L is 34. Considering the number of automaton states and space needed to store the values p_1–p_4 and j, we get approximately 300–500K of memory needed to store all lookup arrays TAB_j, which is quite acceptable for modern computers.

However, the mentioned lookup tables are too large to fit into L1 cache memory and maybe even to L2 cache, which leads to slowing down the algorithm. In order to eliminate the problem, we developed a method of packing the value $TAB_j[Text[i]]$ into a single 32-bit machine word. Apart from reducing the size of lookup tables, it allows us to load all needed values into a processor register via one memory read at each iteration of the decoding loop, requiring, however, some extra bit-level operations to extract them. In total, this method can accelerate the decoding by 15–25%. We don't explain it in detail, since it is based on a lot of low-level specifics, not impacting the Alg. 1 itself. Nevertheless, this approach was implemented in program and results of its testing are shown in the next section.

4. Experimental Results

The discussed data compression codes were tested on two English and two Ukrainian texts of different size. The smaller one is The Bible – English, King James version, and Ukrainian translated by I. Ohienko. The larger Ukrainian text consists of 27 novels by different 19–20th century writers, 15.5MB in size, while the larger English text contains the 100MB collection of articles randomly taken from Wikipedia. The larger English text was preformatted by eliminating punctuation marks; upper- and lowercase letters are distinguished. The Ukrainian Bible was preformatted by eliminating the verses numbers. Two other texts were encoded without preformatting. Let us note that although larger Ukrainian and English texts are different in size, they contain close numbers of different words and thus are comparable from data compression point of view.

The compression efficiency of different codes applied to this corpus is shown in Table 2, together with parameters of texts. The compressed text size is given in bits per word with the exceedance over the entropy shown in percentage. For each text, the best result is highlighted in bold. The parameters of SCDC and RPBC codes for each text have been chosen to maximize the compression ratio: RPBC(129,125,1,0) and SCDC(172) for Ukrainian Bible; RPBC(191,63,1,0) and SCDC(197) for English Bible; RPBC(110,140,5,0) and SCDC(165) for Ukrainian fiction corpus; RPBC(153,97,5,0) and SCDC(175) for English Wikipedia articles.As seen, among investigated codes, different representatives of RMD-family show the best compression ratio for all texts in both languages.

This is confirmed by the graph in Fig. 5, illustrating the Ukrainian and English Bibles encoding. Both text and codeword set distributions are shown. Positions of words ordered by decreasing frequency are shown in x-axis, differing by a factor of 1.2. Y-axis contains the codeword lengths for codeword sets and values -log p_i for texts (p_i is the probability to meet the i-th word in the text). A theoretically best codeword set should reach the entropy limit $H = \sum_i -p_i \log p_i$, i.e. the length of i-th codeword (in the order of increasing lengths) has to be equal –log p_i. Thus, the closer a codeword set curve to a text curve, the better compression ratio a code provides. As seen, both $R_{2,-\infty}$ and $R_{2,-\infty}$ curves align with English text curve (blue) better than with Ukrainian (orange), although $R_{2,4,-\infty}$ is a bit closer to Ukrainian curve than $R_{2,-\infty}$.

In fact, all codes, except for byte-aligned ones, compress English texts better than Ukrainian, since the Ukrainian distribution curve is too flat (that is to say, words in Ukrainian text are distributed more uniformly). At the same time, the `ladder-shaped' distribution curve of byte-aligned RPBC codes is not so far from flatter Ukrainian word distribution than from more steep English. Nonetheless, the compression ratio of byte-aligned codes, in comparison with other codes, remains for Ukrainian texts the worst, although almost on a level with Fibonacci Fib2 and the Remote Multi-Delimiter $R_{4,-\infty}$ codes.

Generally, for Ukrainian texts the dispersion of deviations of compressed text size from entropy is less, while the medium deviation is bigger than for English. This means that some codes in research are suited for English text compression well and some not, while any code cannot be considered as very well suited for Ukrainian natural language texts. However, due to existence of many customizable parameters, reverse multidelimiter codes can be aligned with Ukrainian natural language text word distribution more tightly than other codes of the discussed class.

Table 2
Codes properties and compression ratio

	Ukrainian Bible	Ukrainian fiction corpus	English Bible, King James version	English Formatted Wikipedia Texts
Entropy	11.986	13.436	9.48	11.078
Total words (TW)	586 254	1 428 193	766 112	19 507 784
Different words (DW)	74 996	258 215	28 659	288 179
DW/TW	7.817	5.531	26.732	67.693
Fib2	13.373 (11.6%)	15.267 (13.6%)	9.993 (5.4%)	11.983 (8.2%)
Fib3	12.561 (4.8%)	14.061 (4.7%)	9.844 (3.8%)	11.564 (4.4%)
SCDC	13.858 (15.6%)	15.411 (14.7%)	11.024 (16.3%)	12.869 (16.2%)
RPBC	13.522 (12.8%)	15.035 (11.9%)	10.953 (15.5%)	12.685 (14.5%)
$R_{2-\infty}$	12.81 (6.9%)	14.661 (9.1%)	9.711 (2.4%)	11.598 (4.7%)
$R_{2,4-\infty}$	12.506 (4.3%)	14.117 (5.1%)	9.749 (2.8%)	11.393 (2.8%)
$R_{2,3,5-\infty}$	12.548 (4.7%)	14.29 (6.4%)	9.989 (5.4%)	11.446 (3.3%)
$R_{3-\infty}$	12.539 (4.6%)	14.013 (4.3%)	9.989 (5.4%)	11.514 (3.9%)
$R_{3,5-\infty}$	12.791 (6.7%)	14.221 (5.8%)	10.313 (8.8%)	11.782 (6.4%)
$R_{4-\infty}$	13.206 (10.2%)	14.584 (8.5%)	10.809 (14%)	12.233 (10.4%)

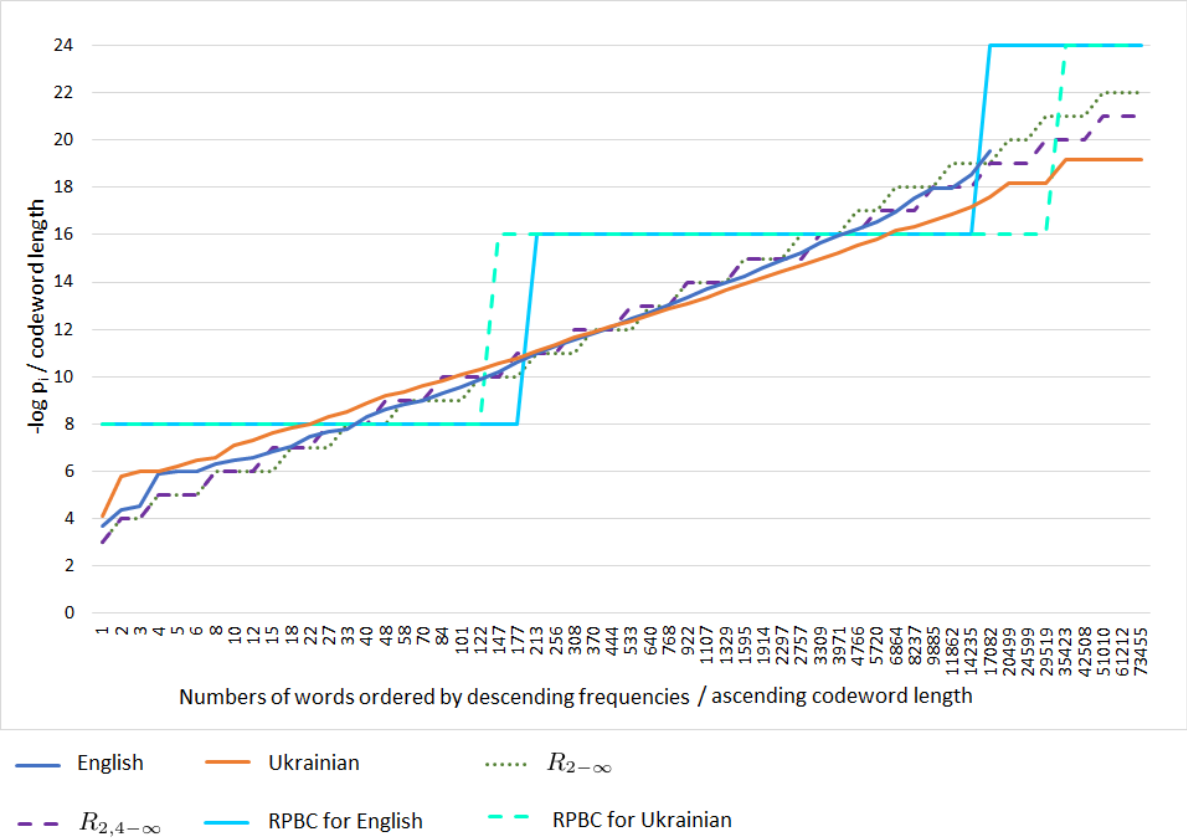

Figure 5: The distribution of text word frequencies and codeword lengths

The average decoding time is given in Table 3. The decoding algorithm was implemented in C programming language and compiled with gcc compiler, -O3 time optimization enabled. The results of decompression were stored in RAM as arrays of decoded numbers. This approach is more indicative than the full decoding with restoring an original text, since converting numbers to strings is quite time consuming and neutralizes the difference between methods performance. Given values represent average time of 100 runs of each algorithm on the PC with AMD Athlon II X2 245 2.9GHz processor, 64K L1 cache, 1MB L2 cache, 4GB RAM, running OS Windows 10.

For each text, to measure the decoding time, we choose RMD-code maximizing the compression ratio of the text, according to Table 2. As seen, the RMD-codes decoding appears to be 40–80% slower than RPBC decoding, 20–30% slower than SCDC decoding and almost 2 times faster than the decoding of Fibonacci code Fib3. It is worth to note that RPBC codes providing the best decoding time are not synchronizable and do not allow the direct search in a compressed file. Also, the results show that decoding time does not depend on language of text or Different words to Total words ratio but depends only on the size of a text and code type.

Table 3
Average decoding time, milliseconds

Code\Text	Ukrainian Bible	Ukrainian fiction corpus	English Bible, King James version	English Formatted Wikipedia Texts
Fib3	11.7	27.8	10.87	281
SCDC	4.56	12.4	4.62	139
RPBC	3.46	11.1	3.47	101
RMD	6.06	15.1	6.25	173

5. Conclusions

The reverse multi-delimiter data compression codes are defined and their application to English and Ukrainian natural language text compression is investigated. These codes are universal, complete, synchronizable and allowing the direct search in a compressed file. Although RMD-codes compress Ukrainian texts less efficiently than English, they provide better compression ratios for both languages than other codes with similar properties. In general, compression of Ukrainian texts appears to be more challenging because of flatter (i.e. more uniform) distribution of word frequencies in Ukrainian. However, since the reverse MD-codes are multiparameterized, they are agile and can be aligned either to English or to Ukrainian texts more tightly than other codes in research. Considering either compression ratio or decoding time, the reverse multi-delimiter codes can be considered as an attractive alternative to byte-aligned and Fibonacci codes in compression of texts in different languages. Construction of codes compressing Ukrainian texts on a level with English and having all mentioned above properties is an unconquered challenge.

6. References

[1] A. Anisimov, I. Zavadskyi, Variable-length prefix codes with multiple delimiters. IEEE Transactions Information Theory 63.5 (2017): 2885–2895.

[2] N. Brisaboa, A. Farina, G. Navarro, 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, 2003, pp. 122–136.

[3] J. Culpepper, A. Moffat, Enhanced byte codes with restricted prefix properties, in: String Processing and Information Retrieval, SPIRE'05, 2005, pp. 1–12.

[4] J. Duda, K. Tahboub, N. Gadgil, E. Delp, The use of asymmetric numeral systems as an accurate replacement for huffman coding, in: Proceedings of Picture Coding Symposium, 2015, pp. 65–69.

[5] D. Huffman, A method for the construction of minimum-redundancy codes, in: Proc. IRE 40, 1952, pp. 1098–1101.

[6] S.T. Klein, M.K. Ben-Nissan, On the usefulness of Fibonacci compression codes. Computer Journal 53.6 (2010): 701–716.

[7] I. Zavadskyi, A. Anisimov, A family of data compression codes with multiple delimiters, in: Proceedings of the Prague Stringology Conference, 2016, pp. 71–84.

[8] P. Elias, Universal codeword sets and representations of the integers. IEEE Transactions Information Theory vol. 21 (1975): 194–203.