CEUR-WS.org/Vol-3384/Short_5.pdf

Practical Word-based Text Compression Using the Reverse Multi-Delimiter Codes

Anatoly V. Anisimov, Igor O. Zavadskyi and Timofey S. Chudakov

Taras Shevchenko National University of Kyiv, 4d Glushkov Ave., Kyiv, Ukraine

Abstract
We present the technique of word-level natural language text compression improving the performance of modern powerful achievers, such as 7z or BC-ZIP. It is based on the use of the Reverse Multi-Delimiter codes in combination with special preprocessing of a text and its dictionary. Also, we construct a very fast decoding algorithm for RMD-codes operating almost at the same speed as SCDC and times faster than Fibonacci codes decoding.

Keywords [1]
Word-based, compression, archiver, code, multi-delimiter

1. Introduction

Compression of large textual databases is one of the key elements of modern information retrieval systems. Text compression techniques can be divided into 2 groups: methods operating individual characters as alphabet elements and methods using words as atomic symbols. We focus on methods of the second group since they rely on partitioning texts (at least in European languages) in most natural semantic units and thus provide significantly better compression ratios. Well-known classical solutions based on entropy encoding, such as Huffman codes [1], can be applied to the word-level text compression and provide compression ratios close to the theoretical limit defined by Shannon's entropy. However, not only the compression ratio matters but also such features as fast search in compressed data, high decoding speed, and code robustness in the sense of limiting possible error propagation. As is known, Huffman codes are not well suited for such requirements.

An alternative approach stems from the use of variable-length codes with delimiters, such as Fibonacci [2], (*s*,*c*)-dense [3], Multi-Delimiter, or Reverse Multi-Delimiter (RMD) [4] codes. Delimiters are special bit sequences denoting the beginning or the end of a codeword. Properly chosen delimiters guarantee that any portion of an encoded bitstream can be uniquely decoded independently of the surrounding context. This implies such remarkable property of a code as synchronizability and also allows us to make the fast Boyer-Moore-style pattern search in a compressed file.

Of course, these properties are achieved at the cost of compression ratio. When we use a character-based alphabet, the price is high enough. However, a word-based alphabet contains rather more elements and distribution of their frequencies is flatter, which equates the compression performance of different codes. As shown in [4], 100 MB English text file compressed by SCDC on a word alphabet may exceed Shannon's H0 entropy by 14.5%, Fibonacci code Fib3 – by 4.4%, and RMD-codes – by 2.8%. Let us note that the mentioned compression ratios characterize codes themselves, without auxiliary structures, such as dictionaries required for encoding and decoding. In contrast to character-level, in a word-level text compression, a dictionary is a more significant part that should be stored together with the compressed file. It can be represented as the sequence of pairs (word of a text, codeword), where codewords are sorted in ascending order of their lengths, while words are sorted in descending order of their frequencies. However, for monotonous codes (e.g. SCDC, Fibonacci, and

Information Technology and Implementation (IT&I-2022), November 30 - December 02, 2022, Kyiv, Ukraine
EMAIL: a.v.anisimov@knu.ua (A. 1); ihorzavadskyi@knu.ua (A. 2); timofey.chudakov@gmail.com (A. 3)
ORCID: 0000-0002-4826-5265 (A. 1); 0000-0002-1467-2006 (A. 2); 0000-0002-1050-1226 (A. 3)
© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

RMD but not MD-codes), it is enough to store only the sequence of different words of a text, since for a given word its codeword can be easily calculated from the word number and vice versa.

The uncompressed word-level dictionary for 1 GB English text occupies about 2.5–3% of a text itself and about 5% of 100 MB text. If we compare the size of a compressed dictionary with the compressed text, the percentage becomes even higher. By the other hand, a text already consists of all dictionary elements, giving us an idea of saving the space by special marking dictionary elements in the text. Also, some other regularities of natural language word sequences can be exploited. This leads us to constructing a word-level text preprocessor, that is placed in front of a standard postcompressor to achieve a higher compression ratio. We describe such preprocessor in Section 2. Although it can be postcompressed by any monotonous variable length code, in experiments we use RMD-codes as they provide the best compression ratio among other codes with delimiters. Notably, the described preprocessing technique preserves the synchronizability of a code, the possibility of fast decompressing, and, to some extent, of a compressed search. Also, it is worth to note that this approach improves not only the performance of variable-length codes but also of known powerful archivers if we apply them after preprocessing + RMD encoding. Moreover, using RMD-codes to encode the lengths and offsets of segments in the LZ77 compression scheme can improve the results of high-order character-based entropy compression, as shown in Section 6 on the example of the open-source archiver BC-ZIP [5].

As mentioned above, another important property of compression codes is a decoding speed. The (s,c)-dense codes are specially intended for fast decoding and considered a champion solution by this parameter, if not considering the recently invented Binary Coded Ternary encoding [6] and Byte Codes with Restricted Prefix Properties [7], although the latter do not possess other properties of codes with delimiters. In Section 3 we suggest a very fast decoding method for RMD-codes operating almost at the same speed as SCDC decoding and even faster for short texts. It essentially improves our previous decoding technique given in [4]. Let us define an RMD-code. Assume $m_1, \dots, m_t$ is the ascending sequence of natural numbers. The codeword set of the RMD-code $R_{m_1,\dots,m_t}$ consists of codewords of the form $01^{m_i}, i = \overline{1,t}$, and also codewords that:

- start from the sequence $01^{m_i}0, i = \overline{1,t}$ and do not contain any of these sequences anywhere else in a codeword;
- do not end with a sequence $01^{m_i}, i = \overline{1,t}$.

The bit sequences of the form $01^{m_i}0$ can be considered as delimiters. Although such delimiters do not belong to codewords of the form 01^{m_i}, these codewords constitute the delimiters together with the leading 0 bit of the next codeword.

In this paper, we use only RMD-codes with the infinite number of delimiters as they demonstrate better compression ratio [4]. By $R_{m_1,\dots,m_t-\infty}$ we denote the RMD-code having the delimiters with $m_1, \dots, m_t$ or greater number of ones. E.g. $R_{2,4-\infty}$ is the code with delimiters 0110 and 01^t0, where $t \geq 4$, while $R_{2-\infty}$ is the code with all delimiters consisting of 2 or more ones.

2. Text preprocessing

As usual, in practical natural language text compression, a preprocessor is placed in front of a standard postcompressor to achieve a higher compression ratio. The most known preprocessors, such as WRT [8], StarNT [9], LIPT [10], and others use a character-based alphabet and exploit such regularities as q-gram compression, treating uppercase letters as lowercase, separate compression of typical word suffixes and roots, etc. We have built our own preprocessor for word-based alphabets. It assumes storing the dictionary together with the encoded text, postcompressing a document with RMD-codes, and after that, applying some powerful achievers, such as 7zip.

2.1. Low frequency word collapsing

Let us give some considerations on redundancy related to words with frequency 1. In fact, each word we store twice: the first time in the dictionary and the second time as a codeword in the encoded text. We could avoid storing a codeword by replacing it with an original word in the text but then additional issues with marking the beginning and the end of a word arise. Therefore, we suggest the following scheme:

1. Sort the dictionary part consisting of the words of frequency 1 in the order they occur in the text.
2. Encode all words of frequency 1 in the text with the same short codeword, say $code_1$.
3. While decoding, replace each $code_1$ with the next item from that part of the dictionary.

As a result, to store each *word* of frequency 1 we use $|word|+|code_1|$ bits instead of $|word|+|code(word)|$ bits. The frequency of the codeword $code_1$ is equal to the number of words of frequency 1, which defines its position in the dictionary. Thus, we get $|code_1| \le |code(word)|$, and the described technique can save space. This technique can be generalized to words of frequency more than 1. During encoding, when we meet a word of frequency $k>1$ first time, replace it by the codeword $code_k$, and encode the next occurrences of this word with its original codeword.

2.2. Word capitalization

This transformation operates as follows. First, a new placeholder is added to the dictionary. This placeholder will serve as an indication of an anomaly in the text. Then all consecutive pairs of words are processed. If the first word in a pair ends with the dot character '.', possibly, it is the end of the sentence. In this case, we check if the first letter of the second word is in uppercase. If so, the frequency of the lower-cased word is compared with the frequency of its unmodified version. If the lower-cased version has a higher frequency, the word is substituted with the lower-cased version. Otherwise, no action is taken. Consider what happens when the second word starts from a lowercase letter. The decoder won't be able to make a correct decision just by looking at the word itself. Then the anomaly placeholder is inserted before this word to make the text uniquely decodable. The first character of the second word in the pair may be neither in lowercase nor in uppercase. If so, the encoder takes no action.

During the decoding phase, we check all consecutive pairs of words in the text and determine if such a pair is a sentence transition. If so, the second word is checked if it is the anomaly placeholder. If so, this placeholder is removed from the text. If not, it is checked if the first character of the second word is a lowercase letter. If it holds true, it is converted to uppercase, otherwise, no action is taken. Note that this transformation may convert all occurrences of some word to the lowercase. Then the dictionary may contain a word with zero frequency. This word might be removed from the dictionary later.

2.3. Local dictionary

Consider all words of the text which follow some chosen word. This subset of the text we denote by S and call a *local dictionary*, while the seed-word we call a *sentinel* word. The word frequency distribution in the local dictionary differs from the global one. It is possible to decrease the global text entropy by choosing sentinels among high-frequency text words and cleverly recoding local dictionaries. Namely, form the set S_L of size L consisting of codes of elements of S with the highest local frequencies. After the sentinel word, the elements of S_L are encoded with codewords $code(0),\ldots, code(L–1)$. All remaining words from $S\backslash S_L$ are encoded as follows: if the position of a word in the global dictionary is less than L, a special *anomaly code* is inserted behind its codeword, otherwise, the codeword is left intact. The anomaly code must be chosen depending on the frequency of the anomaly which can be either less or greater than L. The decoding is done similarly to the encoding. For a given codeword $code(x)$ after the sentinel word, if $x<L$, $code(x)$ is mapped back. If it is equal to the anomaly code, it's deleted, otherwise, the code is left intact.

It is possible to use several local dictionaries in parallel, one for each sentinel word. Each local dictionary together with its sentinel word, the parameter L, and the anomaly code should be stored as a part of a compressed binary sequence. Of course, as the frequency of a sentinel word is higher, the local dictionary becomes more cost-efficient. And, for a given local dictionary, there exists some optimal value of L, when its further increasing leads to many anomaly codes. Therefore, the number of local dictionaries and the value L for each dictionary have to be balanced.

2.4. Using *q*-grams

Although the difference between q-order and 0-order entropy for word-based alphabets is rather smaller than for character-based alphabets, a special encoding of some frequent q-grams can reduce the total bit length of a compressed file. Assume the dictionary is divided into parts corresponding to the codewords of the same length. For RMD-codes, the lengths of codewords of each next part are one bit longer than that of the previous part. Consider a q-gram of words and denote its frequency in the text as l. This frequency corresponds to some codeword c. We can replace each series of codewords corresponding to this q-gram with the codeword c.

Let $|c|$ be the length of this codeword and C be the total length of codewords it replaces. If the dictionary is already composed, inserting a new codeword shifts the rest of the dictionary. Let $q_1,\dots,q_m$ be frequencies of last words in parts of the dictionary containing codewords of lengths $|c|,|c|+1,\dots,|c|+m$. After inserting, lengths of their codewords will be increased by 1, which adds $q_1+\dots+q_m$ extra bits to the encoded text. Then the gain from encoding the q-gram with one codeword is $l(C-|c|) - q_1-\dots- q_m-a$, where a is the size of the archived record relating to that q-gram in the dictionary. If this value is positive, replacing of the q-gram makes sense.

3. Fast byte-aligned decoding

Any Reverse Multi-Delimiter code can be considered as a regular language and thus recognized by the finite automaton. The decoding automata for codes $R_{2-\infty}$, $R_{3-\infty}$, and $R_{2,4-\infty}$ are given and discussed in [12]. However, they process a text bit-by-bit, which is quite slow. The main idea of a fast decoding algorithm is a "quantification" of a decoding automaton so that it reads bytes of a code and produces the corresponding output numbers.

We use the following notations. Assume we have finished processing some byte of a code. The *pointer ptr* is a combination of the decoding automaton state a and the number l of already decoded bits of the last codeword. It can be calculated as follows: $ptr = a\cdot l_{max}+ l$, where l_{max} is the maximal possible bitlength of a codeword. If we multiply *ptr* by 256 and add a current byte of the text, we get the index x of lookup tables (line 3 of Algorithm 1). We utilize the following lookup tables:

- *Pointers*[x] – the pointer for decoding the next byte;
- *Numbers*[x] – a 64-bit number, which consists of four 16-bit numbers we get after decoding a current byte;
- $c[x]$ – the number of codewords fully decoded during processing the current byte.

At first, we describe a fast decoding algorithm for RMD-codes assuming the dictionary contains no more than 2^{16} words (Algorithm 1). In this case, 4 sequential decoded integers can be output with one assignment of a 64-bit value. On the other hand, it is easy to show that no more than 4 codewords can be decoded, fully or partially, while processing one byte of a code with the shortest delimiter 011. Combining these two facts, we get Algorithm 1 for decoding texts with a short dictionary. The loop iterating over bytes of the code is given in lines 2–8. In lines 3 and 4 we calculate the pointer given the current byte of the code *Code*[i] and the previous value of the pointer *ptr*. Then, in line 5, we output 4 decoded numbers, even if the actual number of outputs produced by the current byte is less. In line 6 the current output position is shifted by this actual number of outputs. This means that some elements of the array *Out* may be overwritten at next iterations of the decoding loop.

```
Algorithm 1: Byte-aligned decoding of the RMD-code for short texts
   input : RMD-bitstream, composed of bytes, $Code[1 \dots n]$.
   output: Array of numbers $Out$.
1  $ptr \leftarrow 0$
2  for $i \leftarrow 1$ to $n$ do
3      $x \leftarrow ptr * 256 + Code[i]$
4      $ptr \leftarrow Pointers[x]$
5      $Out[k, \dots, k+3] \leftarrow Numbers[x] + tr$
6      $k \leftarrow k + c[x]$
7      $tr \leftarrow Out[k]$
8  end
```

The byte-aligned decoding is illustrated in Fig. 1. During decoding a current byte, the last codeword is decoded partially (e.g., the codeword k+2 in Fig. 1). If the first decoded codeword is stored in the element *Out*[*k*], the mentioned last codeword is stored in the element *Out*[*k*+*c*[*x*]]. Thus, the result of its partial decoding is assigned to the variable *tr* in line 7. On a little endian machine, this value is added to the first decoded number at the next iteration of the decoding loop in line 5, since bytes of a value are loaded from memory to a processor register and vice versa in the reverse order. Execution of line 5 of Algorithm 1 on a little endian machine is illustrated in Fig. 2. Note that on a big endian machine an extra operation of shifting the value *tr* to the left on 48 bits is needed in line 5.

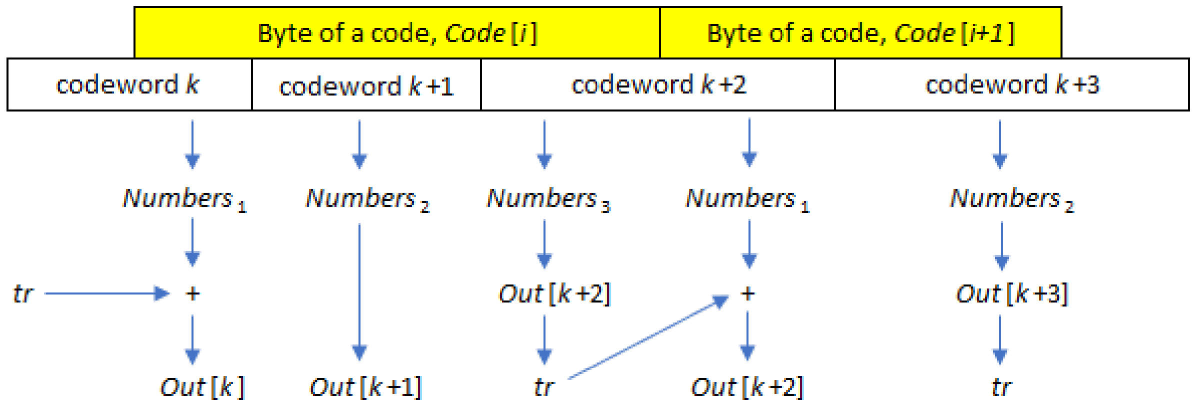

Figure 1: Decoding two bytes of an RMD-code

Note that, unlike the fast decoding algorithm for RMD-codes presented in [4], we do not analyze the number of codewords to be decoded at the current iteration of a decoded loop. This is because the 'if' statements used for that purpose are unpredictable (i.e. can be either 'true' or 'false' with high probability), and executing such unpredictable 'ifs' is one of the main reasons for slowing down the programs on modern processors. The general idea of the described above algorithm resembles the fast decoding algorithm for the BCT-code presented in [6]. However, the BCT lookup table index does not depend on the length of the already decoded part of a codeword. Therefore, that table is rather more compact, and the BCT-code can be decoded essentially faster, though providing a worse compression ratio. If the dictionary contains more than 2^{16} words, line 5 of Algorithm 1 should be replaced with the following two lines:

$$\begin{aligned} Out[k, k+1] &\leftarrow Numbers[x] + tr \\ Out[k+2, k+3] &\leftarrow Extranumbers[x] \end{aligned} \tag{1}$$

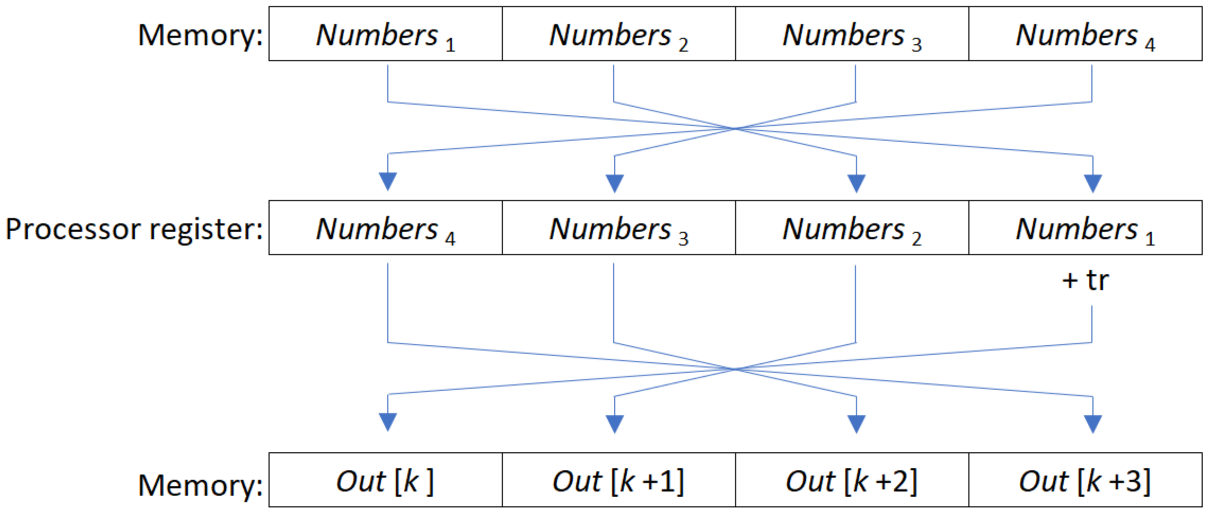

Figure 2: Execution of line 5 of Algorithm 1 on a little endian machine

Here *Extranumbers* is the special array with 64-bit elements containing third and fourth numbers that may be obtained during decoding a current byte. This allows us to process texts with a dictionary containing up to 2^{32} elements, which covers all practically interesting cases. Let us estimate the space overhead posed by Algorithm 1. Assume we decode $R_{2-\infty}$ having 3 states of the decoding automaton. The length of a codeword in the dictionary with no more than 2^{16} words does not exceed 24. This gives $3 \cdot 24=72$ possible values of *ptr* and $72 \cdot 256=18432$ possible values of x obtained in line 3. Each element of the array *Numbers* occupies 8 bytes, the array *Pointers* – 1 byte, and the array c also 1 byte. Thus, all lookup tables occupy 184 KB, which fits into the L2 cache on most machines. For long texts, the size will be about twice as large, and this comprises a typical L3 cache size.

4. Experiments

We tested how the described above text preprocessing technique can improve the compression efficiency of RMD-codes and popular archivers. The experiments were conducted for the 1GB text from the Pizza&Chilie corpus. The original text consists of 1,073,741,824 bytes, 189,528,100 words, 2,523,827 unique words. The file word-level entropy is 273,284,721 bytes, 13.535 bits per word. The results are shown in Tables 1 and 2.

In Table 1 heads of columns denote applied preprocessing actions.

(1) Encoding + Dictionary.

(2) Encoding + Dictionary + Low frequency word collapsing.

(3) Encoding + Dictionary + Low frequency word collapsing + Word capitalization.

(4) Encoding + Dictionary + Low frequency word collapsing + Word capitalization + Local dictionary. We use local dictionaries for 60 most frequent sentinel words and each local dictionary consists of 230 elements.

(5) Encoding + Dictionary + Low frequency word collapsing + Word capitalization + Local dictionary + q-grams. We include only q-grams reducing the encoded file size. Usually, including only frequent trigrams appears to be most efficient.

As seen, applying local dictionaries gives the most effect, while the special encoding of q-grams after all other transformations helps a little. Results of archiving are given in Table 2. We applied 7z, version 16.02, 64-bit, RAR, and gzip archivers in the maximum compression mode (level 9) to the original and RMD-encoded texts with all mentioned above preprocessing transformations.

Table 1
Compression with preprocessing

Code	(1)	(2)	(3)	(4)	(5)
$R_{2-\infty}$	310,919,900	307,355,686	304,738,038	295,795,588	295,561,517
$R_{2,4-\infty}$	305,141,686	301,223,784	298,519,942	289,765,845	289,577,183
$R_{3-\infty}$	307,282,575	303,666,323	301,172,665	293,240,306	292,954,265

Table 2
Compression with preprocessing, RMD-encoding and further archiving

Archiver	Original text	$R_{2-\infty}$	$R_{2,4-\infty}$	$R_{3-\infty}$
7z	258,428,183	258,705,282	257,252,378	256,751,472
RAR	290,584,311	264,645,314	261,948,637	262,563,336
gzip	405,714,638	277,592,303	273,990,536	274,736,358

As observed, preliminary encoding with RMD-codes significantly improves RAR- or gzip-compression ratio, by more than 10% or almost 50% respectively. LZMA-based 7z compresses the text much better than RAR or gzip and it is recognized as one of the most powerful modern archivers. However, even in this case RMD-codes open room for improvements. For example, the 7z-archiving of the $R_{3-\infty}$-encoded text produces 0.7% smaller file than archiving this text without the RMD-preprocessing. Let us note that preprocessing transformations preserve the synchronizability of a code as well as the possibility of fast decoding. After all transformations, except for the low frequency word collapsing, the fast Boyer-Moore- style pattern search also remains possible but requires, however, to process special cases. Also, it is interesting that not archived RMD-encoded files are 32–34% smaller than files archived with gzip. Considering the possibility of compressed search and fast decoding, RMD-codes can be considered as a preferred format to store large textual databases compared with gzip, which decodes texts 3–4 times slower. We measured the decoding time on the AMD Athlon 3000G processor, 32 KB of L1 cache, 512 KB of L2 cache, 4 MB of L3 cache, 16 GB RAM, and 64-bit operating system Windows 10. Results are shown in Table 4 for 3 texts of different size:

- Small: The Bible, King James version, 3.83 MB, 790 018 words in total, 14 087 distinct words, SCDC(224,32).
- Middle-sized: articles randomly taken from Wikipedia (116 MB, 19 507 783 words in total, 288 179 distinct words), SCDC(175,81).
- Large: the first half of the largest file from the Pizza&Chili Corpus, (512MB, 92 424 896 words in total, 1 686 371 different words), SCDC(164,92).

For comparison, we chose other known codes, that make a compressed search and/or fast decoding possible: the Fibonacci codes Fib2 and Fib3 and the byte-aligned (s,c)-dense codes. SCDC are parameterized, and for them we chose the code parameters providing the best compression ratio for each text. Texts and (s,c)-code parameters are indicated below. For SCDC it is a pair (s,c).

As seen, RMD-codes can be decoded almost as fast as SCDC and times faster than the Fibonacci code Fib3. For the short text, the special decoding algorithm (Algorithm 1) operates even faster than SCDC. For longer texts, Algorithm 1 can be applied only with amendment (1) and it is 4–11% slower than SCDC-decoding. Also, as seen, the $R_{2-\infty}$-code can be decoded 4–7% faster than $R_{2,4-\infty}$. This can be explained by the larger lookup tables of the latter code (the decoding automaton for $R_{2,4-\infty}$ consists of 5 states vs 3 states for $R_{2-\infty}$). Also, experiments on pattern matching in encoded texts have been conducted. Both SCDC and RMD-codes allow fast pattern search of the Boyer-Moore type in the encoded file without its decompression. Generally, the pattern search in an SCDC-file can be performed on the byte level, while in an RMD-file we should analyze the values of bit patterns. This means that the pattern search in an SCDC-file should be faster. However, the fastest known up-to-date family of bit-pattern search methods [11] performs the bit-pattern search on the byte level, and only when a candidate substring is found, the individual bits are analyzed. This allows performing the bit-pattern search at a speed comparable to the byte-pattern search.

Table 4

Empirical comparison of decoding time (milliseconds)

Text	Fib3	SCDC	$R_{2-\infty}$ – short	$R_{2-\infty}$ – long	$R_{2,4-\infty}$ – long
Small	7.38	2.73	2.69	3.08	-
Middle-sized	240.3	96.6	-	100.6	108
Large	984	479	-	507	525

We apply one of such methods described in [13] to search for a pattern in the mentioned above Large text. For comparison, the text was encoded both with SCDC and $R_{2,4-\infty}$-codes. RZ*k*Byte-w*n* pattern matching algorithms was applied to the SCDC-code and RZ*k*Bit-w*n* algorithms to the RMD-code, where *k* is the number of significant bits in a mask and *n* is the number of sliding windows. Different values of parameters *k* and *n* have been tested to find all occurrences of encoded sequences of 2–1024 words randomly taken from the text. For each pattern length, the values (*k*,*n*) giving the best time in average were chosen. The average times of 1000 runs of search algorithms are shown in Table 5, with method parameters (*k*,*n*) in brackets.

Table 5

Empirical comparison of pattern search time (milliseconds)

Code / Pattern length (words)	2	4	8	16	32	64	128	256	512	1024
SCDC	21.38 (12,5)	17.46 (12,5)	17.53 (16,3)	17.02 (16,3)	11.87 (16,3)	6.23 (16,3)	3.78 (16,3)	2.68 (16,3)	2.11 (16,3)	1.61 (15,5)
$R_{2,4-\infty}$	120.91 (15,2)	37.54 (16,1)	20.6 (14,2)	16.73 (16,2)	15.88 (15,2)	15.3 (16,2)	8.81 (16,1)	4.74 (16,2)	2.5 (16,1)	1.67 (16,1)

As seen, for very short patterns (2 words in length), the pattern search in the SCDC-encoded text is 6 times faster. This is because patterns of lengths <16 bits cannot be searched on the byte level. For longer patterns of some lengths (4, 64, or 128 words) SCDC-search is more than twice faster. However, there are some commonly searched pattern lengths when the RMD-search is almost on the same level as the SCDC-search or even a little faster (8 or 16 words). As a pattern becomes longer, the difference between performances of bit and byte pattern matching methods goes to zero, which is demonstrated by the results for patterns of 512 and 1024 words in length.

5. Reverse multi-delimiter codes in LZ77 character-based compression

The variable-length integer codes are in the heart of character-based data compression schemes. The RMD-codes belong to this class and thus they can be used not only as a preprocessing tool for text compression but also within character-based compression algorithms. For instance, they can encode lengths and distances of segments in the LZ77-style compression. One of the most efficient LZ77-based archivers for the natural language texts is BC-ZIP [12], [13]. Among versions of BC-ZIP, the 'bit-optimal' algorithm provides the best compression ratio. Its core idea is to optimize the LZ-parsing considering the lengths of segment bit representations. Special variable-length codes ('SODA09' and 'Nibble4'), with different parameters for offsets and lengths of LZ-segments, have been developed and implemented in the source code of BC-ZIP [5]. These codes use the Elias gamma-style encoding. A codeword is divided into 2 parts: the first part contains the number of a codeword class given in the unary numeral system, while the second part consists of the main binary sequence. All codewords of the same length comprise a codeword class. In gamma encoding, selecting the proper length classes allows for achieving rather better compression ratios than for other variable-length codes.

Although BC-ZIP codes parameters were selected very carefully to maximize the ratio of the bit-optimal compression, experiments demonstrate that replacing the original BC-ZIP codes with multi-delimiter codes can better compress short texts of different nature. Table 6 shows the results of the bit-optimal BC-ZIP-compression of different texts from the Canterbury corpus and parts of the English text from the Pizza&Chili corpus [14]. Code parameters giving the best compression ratio were chosen both for RMD- and BC-ZIP families. Segment offsets were encoded by $R_{2-\infty}$, $R_{3-\infty}$, or $R_{2,4-\infty}$ codes. Segment lengths are shorter and the best choice for them is a code with the shortest codeword of length 2: $R_{1-\infty}$ or $R_{1-5,7-\infty}$.

As seen, using RMD-codes in the BC-ZIP scheme instead of SODA09 or Nibble4 code improves the compression ratio for all files from the Canterbury corpus, except the grammar.lsp. The compression ratios for the 1MB English text are almost the same. However, when files become larger, the efficiency of RMD-codes decreases since Elias gamma-style codes are better suited for large codeword sets. Nonetheless, compressing short text messages is important, because even large texts are often divided into small windows during the compression (see e.g. the Brotli compressor [15]).

Table 6
Using RMD-codes in BC-ZIP compressor

	Original file (KB)	RMD (KB)	RMD-code (type)	BC-ZIP (KB)	BC-ZIP (type)	RMD out-performance
alice29.txt	152.089	55.362	$R_{3-\infty}$; $R_{1-\infty}$	57.708	Soda-09-8	4.07%
asyoulik.txt	125.179	50.935	$R_{3-\infty}$; $R_{1-\infty}$	53.916	Soda-09-8	5.53%
kennedy.xls	1.029.744	313.505	$R_{3-\infty}$; $R_{1-\infty}$	327.716	Nibble4-8	4.34%
cp.html	24.603	9.629	$R_{2,4-\infty}$; $R_{1-\infty}$	9.894	Nibble4-8	6.32%
fields.c	11.150	3.690	$R_{2-\infty}$; $R_{1-\infty}$	3.761	Nibble4-8	1.89%
grammar.lsp	3.721	1.621	$R_{2-\infty}$; $R_{1-\infty}$	1.582	Nibble4-8	-2.46%
lcet10.txt	426.754	135.819	$R_{3-\infty}$; $R_{1-\infty}$	137.761	Soda-09-8	1.41%
plrabn12.txt	481.161	191.758	$R_{3-\infty}$; $R_{1-5,7-\infty}$	191.847	Soda-09-8	0.05%
ptt5	513.216	64.656	$R_{2-\infty}$; $R_{1-\infty}$	65.633	Nibble4-8	1.49%
sum	38.240	15.346	$R_{2-\infty}$; $R_{1-\infty}$	16.406	Nibble4-8	6.46%
xargs.1	4.227	2.208	$R_{2-\infty}$; $R_{1-\infty}$	2.218	Nibble4-8	0.48%
english.1MB	1000	337.118	$R_{3-\infty}$; $R_{1-\infty}$	337.206	Soda-09-8	0.03%
english.2MB	2144.059	738.917	$R_{3-\infty}$; $R_{1-\infty}$	730.255	Soda-09-8	-1.19%

On the other hand, the bit-optimal LZ-parsing is performed times faster for 'stepped' gamma-style codes. This is because less number of possible codeword lengths require fewer search resources. However, BC-zipped file decompression can be done roughly at the same speed both for gamma-style and RMD-encodings due to the fast decoding algorithm described in Section 3.

6. Conclusion

The reverse multi-delimiter (RMD) compression codes are of special interest. They can be used as a key element for the word-based natural language text compression as well as for the compact representation of unbounded integer sequences. We establish a monotonous invertible mapping between the set of natural numbers and the set of reverse multi-delimiter codewords. This mapping implies the 'decoding-in-parts' principle, allowing us to construct a very fast byte-aligned decoding algorithm based on lookup tables, which is comparable with the (s,c)-dense byte-aligned decoding method. Given a good compression ratio, the RMD-codes provide an attractive point in the trade-off between the compression ratio and the decoding speed in natural language text compression. Together with the special word-level text preprocessing technique, the RMD-codes can serve as a preprocessing tool improving the compression ratio of known archivers. Also, being delimiter-based, the RMD-codes allow using the fast Boyer-Moore style direct pattern search in a compressed bitstream. The experiments show that patterns of different lengths can be searched in SCDC- and RMD-compressed files with a comparable speed.

7. References

[1] D. Huffman, A method for the construction of minimum-redundancy codes, Proc. IRE, vol. 40, 1952, pp. 1098–1101.
[2] A. Apostolico and A. S. Fraenkel, Robust transmission of unbounded strings using Fibonacci representations, IEEE Trans. Inf. Theory, vol. 33, 1987, pp. 238–245.
[3] N. Brisaboa, A. Farina, G. Navarro, and M. Esteller, (s,c)-dense coding: an optimized compression code for natural language text databases, in: Proc. Symposium on String Processing and Information Retrieval, ser. LNCS, no. 2857. SVB, 2003, pp. 122–136.
[4] I. Zavadskyi and A. Anisimov, Reverse multi-delimiter compression codes, in: 2020 Data Compression Conference, 2020, pp. 173–182.
[5] Bc-zip. Open source project. URL: http://farruggia.github.io/bc-zip/.
[6] I. Zavadskyi, Binary-coded ternary number representation in natural language text compression, in: 2022 Data Compression Conference, 2022, pp. 419–428.
[7] J. Culpepper and A. Moffat, Enhanced byte codes with restricted prefix properties, in: String Processing and Information Retrieval, 12th International Conference Proceedings, ser. LNCS, vol. 3772, 2005, pp. 1–12.
[8] P. Skibinski, S. Grabowski, and S. Deorowicz, Revisiting dictionary-based compression, Software, Practice & Experience, vol. 35, no. 15, 2005, pp. 1455—-1476.
[9] W. Sun, A. Mukherjee, and N. Zhang, A dictionary-based multi-corpora text compression system, in: 2003 IEEE Data Compression Conference, 2003, p. 448.
[10] F. Awan and A. Mukherjee, Lipt: A lossless text transform to improve compression, in: 2001 IEEE Inf. Technology: Coding and Computing Conference, 2001, pp. 452–460.
[11] I. Zavadskyi, Fast exact pattern matching by the means of a character bit representation, Springer Nature Computer Science, vol. 3, no. 181, 2022, pp. 1—20.
[12] P. Ferragina, I. Nitto, and R. Venturini, On the bit-complexity of Lempel-Ziv compression, SIAM Journal on Computing (SICOMP), vol. 42, no. 4, 2013, pp. 1521–1541.
[13] A. Frangioni, P. Ferragina, R. Venturini, and A. Farruggia, Bicriteria data compression, ch. 115, pp. 1582–1595. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611973402.115.
[14] Pizza&chili corpus – English texts, URL: http://pizzachili.dcc.uchile.cl/texts/nlang/.
[15] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, Brotli: A general-purpose data compressor, ACM Transactions Information Systems, vol. 37, no. 1, 2019, pp. 1–30.