Efficient Integer Retrieval from Unordered Compressed Sequences

Igor Zavadskyi

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorzavadskyi@knu.ua

Abstract. We investigate the problem of fast direct access to elements of an integer sequence given in a compressed form. If integers are sorted in ascending order, it can be reduced to performing the 'select' operation on a bitmap, which is very well investigated. We focus on the more general and more complicated case of unordered integer sequence and propose to represent it with the help of variable-length Reverse Multi-Delimiter (RMD) codes. When applied to data compression, these codes combine a good compression ratio with fast decoding. In this paper, another property of RMD codes is researched - the ability of direct access to codewords in the encoded bitstream. We present the method allowing us to extract and decode a codeword from an RMD-bitstream in almost constant time and experiment on its application to natural language text compression. Due to the properties of RMD codes and compactness of auxiliary direct access structures, our method appears to be very space efficient, only a few percent above the Shannon entropy.

1 Introduction

A compressed representation of integer sequences is the key element of different data compression techniques. It is used in frequency-based compression, when alphabet symbols are numbered so that smaller numbers are assigned to more frequent symbols, in the compact representation of suffix trees and arrays, offsets and lengths in LZ77-type compression, to mention a few areas. In most cases, the distribution of integers is biased in the direction of smaller ones. The variable length codes (VLC) provide a simple, space-efficient solution to the problem. However, often not only is compression itself required but also performing different operations on compressed integer sequence, such as sequential decoding or extracting the element with a given index. While VLC are well-suited for sequential decoding, the direct access to elements of a VLC-bitstream is not obvious and straightforward. It requires using auxiliary data structures and/or a special code construction.

Suppose integers are arranged in ascending order, and deltas between them are small enough. In that case, the problem of direct access is reduced to performing the *select* operation on a bitmap, where i-th bit is set if the number i belongs to the sequence (as usual, we denote by $select(B, i)$ the position of the i-th one in the bitstream B, while $rank(B, i)$ is equal to the number of ones from the beginning of the bitstream B up to position i). There are a lot of solutions concerning rank and select on bitmaps; the overview of recent results can be found in [13] and [15]. Less attention was paid to extracting an element of an unordered number sequence given in a compressed form. However, this operation is quite useful, e.g., in manipulating compressed texts or calculating values of the Ψ function used in compressed suffix

Igor Zavadskyi: Efficient Integer Retrieval from Unordered Compressed Sequences, pp. 83–96.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

arrays. Construction of data structures for space-efficient representation of unordered integer sequences allowing fast access to their elements is the goal of this paper.

Generally, we can distinguish five approaches to solve the mentioned problem more or less efficiently in terms of space and time. Here and below, we denote by n the number of elements in the integer sequence and assume the RAM model is used, allowing constant time reading values from memory.

1. Encode the sequence using universal codes, such as Elias codes [8]. Sample each h-th codeword and store pointers to sampled elements. To get the x-th element, obtain the position of the $\lfloor x/h \rfloor$-th sampled element and then search the bitstream sequentially. This approach is quite straightforward and requires $O(h)$ time to access an element.

2. Encode numbers with all binary strings of the shortest possible length and concatenate these codes. Construct an auxiliary bit sequence of the same length as the main bitstream denoting by '1' the starting positions of codewords. Then a given codeword can be extracted in constant time via known 'select' techniques applied to an auxiliary bit sequence. This method is discussed in [10] and called *Simple Dense Coding* (SDC). Its drawback is obvious: although the main bitstream can be quite succinct, the auxiliary bitstream doubles the space, which is more than significant.

3. In a dense sampling scheme [9] sequence elements are also encoded with all possible binary strings constituting non-uniquely decodable but very dense code. To access the elements, two types of pointers are stored: absolute pointers to every $O(\log n)$-th element and relative pointers to the rest. If the integers distribution is not too positively skewed, the constant time direct access is possible at the cost of less extra space to the main bitstream compared with the previous approach.

4. Split binary representations of integers into chunks of a fixed length. Construct the first bitstream from all first chunks, the second bitstream from all second chunks, etc. Then the i-th element of any bitstream can be accessed on the RAM model in constant time as an array element. However, since the number of chunks in the bit representation of an integer is variable, an extra bitmap B_j is used together with the j-th bitstream. $B_j[i] = 1$ iff the j-th chunk of some integer is stored in the i-th element of the j-th bitstream, and this integer has the $(j + 1)$-th chunk. Thus, to reconstruct the i-th integer, we directly get its first chunk from the first bitstream and then check the $B_1[i]$. If it is 0, we've got the result; otherwise, we compute the $rank(B_1, i)$ to get the position of the integer's second chunk in the second bitstream and so on. This technique is investigated in [6] and called the *Directly Addressable variable-length Codes* (DAC). Also, it is known as *Reordered Vbyte* since it generalizes the Vbyte code idea [17]. Extraction of a random codeword requires at most $\lceil \frac{\log S}{b} \rceil$ rank operations, where S is the largest integer in the sequence and b is the chunk size. The space overhead for all rank data structures is $O(\frac{n \log \log n}{\log n})$.

5. Use the variable-length codes with delimiters, for example, Fibonacci codes [3], as proposed in another schema from [10]. The classical constant-time and $o(n)$-space 'select' algorithm, proposed by Clark [7], can be extended to this case.

Our method of fast direct access to elements of a compressed integer sequence is based on the use of variable-length Reverse Multi-delimiter (RMD) Codes introduced in [19]. These codes are self-delimited, which allows us to avoid using an auxiliary bit-vector indicating codeword boundaries. As well as for Fibonacci codes, it is easy to extend the classical Jacobson's [12] and Clark's [7] results regarding constant time

rank and select on bit-vectors with $o(n)$ extra space to the case of RMD codes. However, the select operation in [7] requires $\frac{3n}{\lceil \lg \lg n \rceil} + O(n^{\frac{1}{2}} \lg n \lg \lg n)$ bits of extra space. On the real-world data, for bit sequences of length up to 4Gb, this value exceeds 60% of the code itself, which we consider too big. To reduce space overhead, we combined approaches 1 and 5. Namely, we store the absolute position of each 'Level 1' block of RMD-codewords, then use a kind of linear approximation to get the position of a smaller 'Level 2' block (a similar idea has been implemented in [5] and [13]) and search the codeword inside this block sequentially. Properties of RMD codes allow us to perform this search and decoding significantly faster and use smaller space than Elias or Fibonacci codes.

The variable-length data compression RMD codes and their properties are briefly discussed in Section 2. The main method of integer retrieval is presented in Section 3. Its space complexity is estimated in Section 4. In Section 5, we discuss experiments in compression and extracting elements from an integer sequence generated in the process of English text compression. As shown in [19], RMD codes outperform Fibonacci codes in natural language text compression ratio and decoding speed. That is why we did not include in experiments the Fibonacci-based approach [10]. Also, in practical English text compression, DAC outperforms the dense sampling both by space and time, as shown in [6]. Thus, we also exclude the dense sampling scheme from our experimental set, where the SDC encoding, DAC, and our new method remain. Also, as the basis of comparison, we experimented with naive approach 1 implemented using the Elias delta code.

2 Reverse Multi-Delimiter Codes

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

Definition 1 *The reverse multi-delimiter code $R_{m_1,\ldots,m_t}$ consists of all the words of the form $01^{m_i}, i = 1, \ldots, t$ and all other words that meet the following requirements:*

(i) for any $m_i \in \mathcal{M}$ a word does not end with the sequence 01^{m_i};
(ii) for any $m_i \in \mathcal{M}$ a word can contain the sequence $01^{m_i}0$ only as a prefix;
(iii) a word starts with the prefix $01^{m_i}0$ for some $m_i \in \mathcal{M}$.

The given definition implies that code delimiters in $R_{m_1,\ldots,m_t}$ are sequences of the form $01^{m_i}0$. However, the code also contains shorter words of the form 01^{m_i} that form a delimiter together with the first zero of the next codeword. The set of codewords of the code $R_{2,4,5}$ of length not greater than 7 is shown in Fig. 1, where codewords of the second type are highlighted in grey.

In the sequel, we refer to the "infinite" versions of RMD codes, notably $R_{2-\infty}$ and $R_{2,4-\infty}$, as they demonstrate the best compression ratio. They use all delimiters containing $2, 3, \ldots$ or $2, 4, 5, \ldots$ ones, respectively. However, in practice, limiting the lengths of delimiters by some relatively large number is enough, defined by the maximal codeword length for a specific application.

Any reverse multi-delimiter code $R_{m_1,\ldots,m_t}$ contains the same number of codewords of a given length as the "direct" multi-delimiter code $D_{m_1,\ldots,m_t}$ discussed in [1]. Thus, reverse MD-codes possess all properties of MD-codes, such as completeness and universality, as well as their asymptotic densities. For MD codes, these properties were proven in [1]. Also, we refer to [1] for the analysis of asymptotic densities and quantities of short codewords in multi-delimiter codes.

L=3	L=4	L=5	L=6	L=7
011	0110	01100	011000	0110000
		01101	011010	0110100
		01111	011110	0111100
			011001	0110010
			011111	0111110
				0110001
				0110101
				0111101
				0110111

Figure 1. $R_{2,4,5}$ codewords of length ≤ 7

The main advantage of RMD codes over the "direct" multi-delimiter code is that for RMD codes, there exists a simple monotonic encoding mapping from the set of natural numbers to the set of codewords. Also, the reverse decoding mapping can be built and represented as a finite automaton with a small number of states recognizing codeword bits from left to right and calculating the index of a codeword in the ordered codeword set. A simple decoding principle is a key point since, as mentioned above, we must decode the RMD-encoded numbers quickly to retrieve them from a compressed sequence efficiently.

The decoding automata for codes $R_{2-\infty}$, $R_{3-\infty}$, and $R_{2,4-\infty}$ are given and discussed in [20]. 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. Such an algorithm has been proposed in [19] and its improved version in [2]. In this paper, we search for a codeword and, after it is found, decode it. Thus, we need a simplified version of the fast decoding method intended to decode only one codeword. This simplification of the decoding algorithm from [2] we use in Algorithm 2 described in the next Section, which is a part of general integer retrieving Algorithm 1.

3 Integer Retrieving Technique

Below we describe Algorithm 1 calculating the value of the element with a given index in the integer sequence encoded with RMD codes. From now on, we consider codes $R_{2,x}$ having the shortest delimiter 0110; they are the best representatives of an RMD family in natural language text compression. Although an RMD-encoded sequence is a bitstream, we operate on a byte level to make the method fast, getting all required bit-level data from lookup tables. The algorithm idea and notations are the following.

- Split the encoded bitstream into level 1 blocks containing L_1 codewords each. Store the number of the first byte of each block in the array $L1byte$.
- Split each L1-block into level 2 blocks containing L_2 codewords each. Let $L2Length[i]$ be the average length in bytes of an L2-block in the i-th L1-block. Also, store the array $\Delta_b[i][j] = b_{ij} - \lfloor j \cdot L2Length[i] \rfloor$, where b_{ij} is the position of the first byte of the j-th level 2 block relative to i-th level 1 block. Then we can calculate the number of the leftmost byte of the L2-block by the formula $L1byte[i] + j \cdot L2Length[i] + \Delta_b[i][j]$. As shown in [19], no more than three codewords of an RMD-code $R_{2,x}$ can start in one byte. Therefore, we also need the

2-bit value $\Delta_c[i][j]$ indicating which codeword inside the byte is the first codeword of the j-th level 2 block.

– When we know exactly where the L2 block starts, seek the codeword inside the block, processing it byte-by-byte. It can be done from the beginning of the block in the left-to-right direction or from the beginning of the next block right-to-left, depending on which way is shorter.

– When we find the leftmost byte of a required codeword, decode it using a fast byte-aligned decoding technique, e.g., as discussed in [2].

Algorithm 1: Decoding an element of the RMD-encoded sequence

input : The index t of the element.
output: The value of the t-th element, *out*.

1 $n_1 \leftarrow t \operatorname{div} L_1$; `// Number of the L1-block`
2 $e_1 \leftarrow t \bmod L_1$; `// Number of the element inside the L1-block`
3 $n_2 \leftarrow e_1 \operatorname{div} L_2$; `// Number of the L2-block inside the L1-block`
`// Number of the codeword inside the byte-aligned L2-block`
4 $e_2 \leftarrow e_1 \bmod L_2 + \Delta_c[n_1][n_2]$;
`// Find the byte where the codeword starts`
5 **if** $e_2 < L_2/2$ **then**
`// Number of the byte where the L2-block starts`
6 $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2]$;
7 $e \leftarrow 0$;
8 **else**
9 $n_2 \leftarrow n_2 + 1$; `// Search from the next L2-block right to left`
10 $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2] - 1$;
11 $e \leftarrow L_2 + \Delta_c[n_1][n_2 - 1] - \Delta_c[n_1][n_2]$;
12 **while** $e \geq e_2$ **do**
13 $e \leftarrow e - Words(i)$;
14 $i \leftarrow i - 1$;
15 **while** $e < e_2$ **do**
16 $e \leftarrow e + Words(i)$;
17 $i \leftarrow i + 1$;
`// Starting from the` $(i-1)$`-th byte of a code, skip` $e - e_2$ `codewords, and decode the next codeword`
18 $out \leftarrow Decode\_number(i - 1, e - e_2)$;

Note that, in general, the j-th L2-block position can be approximated by the formula $kj + b$, where parameters k and b are calculated with the ordinary least squares technique. However, we intentionally fix b as $L1byte[i]$ since experiments show that this approach is a bit less space-consuming.

Let us explain in detail how Alg. 1 works. Given the index t of a required element (codeword), in lines 1 and 2, we get the number of the containing L1-block, n_1, and the relative number of the element inside this L1-block, e_1. Consider the L2 block containing the required codeword. Its number n_2 relative to the containing L1 block is calculated in line 3 of Alg. 1.

We search a codeword inside the L2-block byte-by-byte. However, an L2 block may start not from the first codeword in the byte but from the second or third. Then we extend the discussed L2-block to the left by including all full codewords

from its first byte and call this extended block a *byte-aligned L2-block*. We store the difference between the codeword positions in the byte-aligned and original L2-blocks in the array Δ_c, and get the number of the required codeword inside the byte-aligned L2-block in line 4 of Alg. 1 denoting it by e_2.

In line 5, we analyze if the codeword is in the left half of the L2 block. If so, using a kind of linear approximation, in line 6, we get the number of the byte where this L2 block starts. Then in lines 7 and 15-17, the number i of the first byte of a required codeword is calculated by sequential processing bytes of L2-block from left to right. The function $Words(i)$ returns the number of codewords starting in the i-th byte of a code. It is summed up in the variable e until it becomes no less than the required value e_2. The correspondence between the estimated and actual beginning of an L2 block, as well as values from Δ_b and Δ_c arrays, are shown in Fig. 2.

If the required codeword is in the right half of the L2 block, the right-to-left search from the beginning of the next L2 block would be faster (lines 9-14). In this case, we increment the number of the L2 block (line 9), get the number of the byte before its beginning (line 10), and the number of codewords the right-to-left search to be started from (line 11). After the search finishes, the value e may become too small, and a few iterations of the left-to-right search may be needed (lines 15-17).

In both cases, after line 17, the index $i-1$ points to the byte where the required codeword starts and it is the $(e-e_2)$-th codeword in this byte if we count from 0 and the right edge of the byte. Thus, we skip $e-e_2$ codewords from the right edge of the byte and return the result of decoding the next codeword to the left of them (line 18). This is done in the function *Decode_number*, which is described in Alg. 2. Its idea resembles the fast byte-aligned decoding method presented in [2].

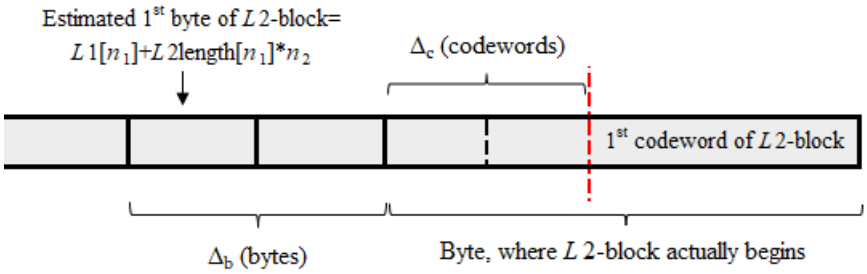

Figure 2. Calculating the position of an L2-block

Codewords for real-world texts are at most 35-40 bits. In line 1 of Alg. 2, we load into the variable *val*64 8 bytes of a code containing the whole codeword to be decoded. In line 2, this codeword is shifted to the right edge of a 64-bit machine word. Then we split the bit representation of *val*64 into chunks and process it chunk-by-chunk, accumulating the resultant value in the variable *out*. Alg. 2 shows this process on a little-endian machine, where bytes of a value are loaded from memory to a processor register in the reverse order. Thus, the bits inside bytes of a code should also be put in the reverse order.

The chunks of an RMD-encoded bitstream are recognized by quantified finite automatons, as described in [19]. The result of the chunk decoding depends on the chunk's content, the number of the chunk, and the state of the decoding automaton at the beginning of chunk processing. The last two parameters are stored in the variable *ptr* used in lines 6 and 7. In line 6, we shift its value by the chunk bit size to the left and add the chunk content to it. This way, we obtain the value v containing the full

information to decode the current chunk. Then, in line 7, we get the value *ptr* for the next chunk and increment the current result by the value $Out[v]$ in line 8. At last, in line 9, the value $val64$ is shifted by the chunk size to the right to process the next chunk. The loop repeats until the flag $N[v]$ signals that we met a delimiter and the codeword has been decoded.

Algorithm 2: Function $Decode\_number(i, s)$ - decoding the $(s+1)$-th codeword starting from the right edge of the i-th byte of a code

input : i - the number of the byte of a code; $s \leq 2$ - the number of codewords to be skipped.
output: The decoded number, *out*.

1 $val64 \leftarrow Code[i...i+7]$; `// Read 8 bytes of a code`
2 $val64 \leftarrow Align(val64, s)$; `// Align the` $(s+1)$`-th codeword to the // right edge of a 64-bit word`
3 $ptr \leftarrow 0$; $out \leftarrow 0$;
4 $chunk\_mask \leftarrow 2^{chunk\_size} - 1$;
5 **repeat**
6 $\quad v \leftarrow ptr << chunk\_size + val64 \& chunk\_mask$;
7 $\quad ptr \leftarrow Pointers[v]$;
8 $\quad out \leftarrow out + Out[v]$;
9 $\quad val64 \leftarrow val64 >> chunk\_size$;
10 **until** $N[v] = 0$;

Example 1. Assume $L_1 = 2^{10} = 1024$, $L_2 = 2^5 = 32$ and retrieve the 1060-th element from the compressed integer sequence encoded by $R_{2,4-\infty}$. At first, we get $n_1 = 1060 \operatorname{div} 1024 = 1$ (the number of the L1-block), $e_1 = 1060 \bmod 1024 = 36$ (the number of the byte in the L1-block), and $n_2 = 36 \operatorname{div} 32 = 1$ (the number of the L2-block). Then, assume the first L1-block occupies 1200 full bytes of an RMD-bitstream, i.e., $L1byte[1] = 1200$, and the average length of an L2-block inside the second L1-block is 40 bytes, i.e., $L2Length[1] = 40$. However, the actual byte length of the first L2 block inside the second L1 block can be different, say 39. Then $\Delta_b[1][1] = 39 - 40 = -1$. E.g., this block can occupy some rightmost bits of the 1200-th byte, full bytes $1201 - 1238$, and five leftmost bits of the byte 1239, as shown in Fig. 3.

Now, assume the binary representation of the 1239-th byte is 00011011. The leftmost 2 bits represent the ending of the 1055-th codeword and, together with the 1056-th codeword 011, belong to the first L2 block inside the second L1 block. Then the last 3 bits 011 are the starting bits of the next L2 block, which interest us. Since this L2-block starts from the second codeword in the byte 1239, $\Delta_c[1][1] = 1$, i.e., we should skip one full codeword in the byte 1239 to get to the beginning of the L2-block. Then $e_2 = e_1 \bmod L_2 + \Delta_c[n_1][n_2] = (36 \bmod 32) + 1 = 5$ is the number of the target codeword if we start counting from the first full codeword in the byte 1239.

Since $e_2 < L_2/2$, we execute lines 6 and 7 of Algorithm 1: $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2] = 1200 + 40 \cdot 1 - 1 = 1239$, $e = 0$. Then we execute iterations of the loop in lines $15-17$ assuming the bitstream is shown in Fig. 3, where even bytes are highlighted with grey.

1. $Words(1239) = 2$, $e = 2$, $i = 1240$;
2. $Words(1240) = 2$, $e = 4$, $i = 1241$;
3. $Words(1241) = 0$, $e = 4$, $i = 1242$;
4. $Words(1242) = 2$, $e = 6$, $i = 1243$;

At last, we call the function $Decode\_number(i-1, e-e_2) = Decode\_number(1242, 1)$. It skips one codeword (1061-th) from the right edge of the byte 1242, aligns the 1060-th codeword 01101 with the right edge of a 64-bit machine word, and returns its decoded value, i.e., 3 (see Fig. 1).

Codeword		1056	1057	1058	1059	1060	1061
Bitstream	0 0	0 1 1	0 1 1 0	0 1 1 1 1	0 1 1 0 1 1 1 0 1 0	0 1 1 0 1	0 1 1 0 1 0 0 0
Byte	1239			1240	1241	1242	1243
L2-block	1st			2nd			

Figure 3. Fragment of an $R_{2,4-\infty}$-bitstream

4 Space complexity

Now, let us estimate the space required by Algorithms 1 and 2, apart from the size of an RMD-encoded file itself. Assume the encoded sequence fits into 4GB, and n is the number of elements in it. Then 4 bytes are enough to store an $L1byte$ array element, or $4n/L_1$ bytes for the whole $L1byte$ array. If we reserve 4 bytes to store the linear approximation ratio $L2Length[n_1]$, the array $L2Length$ will occupy the same space. As mentioned above, no more than three codewords of R_{2-x} code can start in one byte. Therefore, 2 bits are enough for an element of the array Δ_c, or $n/4L_2$ bytes for the whole array.

The function $Word(i)$ uses the lookup table consisting of the number of codewords starting in the byte $code[i]$. Analyzing the byte itself, it is not possible to determine how many codewords start in it. For example, if the byte ends with a 0 bit, it can be either the first bit of the next codeword or a continuation of the current one. However, to answer this question for the code $R_{2-\infty}$, we need to analyze only 2 bits following the current byte and 4 bits for the code $R_{2,4-\infty}$. Namely, the sequences 0|11 in $R_{2-\infty}$, and 0|110 or 0|1111 in $R_{2,4-\infty}$ begin the new codeword, while all other bit combinations after the ending 0 mean that the current codeword continues. For ending 1, we need to analyze even fewer extra bits. Thus, the index of the mentioned lookup table can be a 12-bit integer, and the table consists of 4096 2-bit elements (numbers between 0 and 3). To decrease the number of bit-level operations, we reserve 1 byte for each element, and 4096 bytes will be enough to store the whole table.

To estimate the space complexity of Alg. 2, we should calculate the maximal value of the variable v. If a codeword consists of not more than max_len bits, it contains not more than $c = \lceil \frac{max\_len}{chunk\_size} \rceil$ chunks. As mentioned above, the decoding result depends on the number of the chunk, its content, and the state of the decoding automaton. Thus, $v \leq c \cdot n\_states \cdot 2^{chunk\_size}$, where n_states is the number of states of the decoding automaton (3 for $R_{2-\infty}$ and 5 for $R_{2,4-\infty}$ [20]). Assuming the realistic codeword length does not exceed 35-40 bits and $chunk\_size = 7$, which gives the lowest decoding time in experiments, we get $4000 - 5000$ as an upper bound for v. Each element of the arrays $Pointers$ and N takes 1 byte, while $Out[v]$ requires 4 bytes. Therefore, the total size of the lookup tables for Algorithm 2 does not exceed $25 - 30$KB.

The array Δ_b occupies the biggest space. These delta values can vary in different ranges for different L1 blocks. That is why we allocate the different number of bits

for elements of different $\Delta_b[i]$ subarrays and store these bit lengths in the special array Bit_ranges. We store all Δ_b values for an $L1$-block as a bitstream, keeping a pointer to it in the array Δptr. One byte is enough for a Bit_ranges array element and 4 bytes for a pointer. Of course, this approach involves some extra bit operations. Nonetheless, it allows us to save $40-50\%$ space occupied by data structures needed for direct access and does not affect the overall time much because the biggest time consumption is accounted for the loops in lines $12-17$ of Algorithm 1. Also, using smaller data structures accelerates an algorithm thanks to fewer cache mismatches.

In total, we need $25-30$KB of memory for Alg. 2, $13n/L_1$ bytes for level 1 structures, $n/4L_2$ bytes for the array Δ_c, and a variable space for the array Δ_b. As shown in experiments described in the next section, the optimal value L_1 can be between 2^{14} and 2^{17}, while L_2 is between 2^6 and 2^8. This makes the space for $L1$-structures and Δ_c almost insignificant, about tenths of 1 percent of a code itself, while Δ_b occupies about $1-3\%$ of the code size.

5 Experiments

We tested our solution on integer sequences obtained by applying two known natural language compression schemes to 200 MB English text from Pizza&Chili corpus.

- In the first scheme, words of the text are considered as alphabet symbols. In the dictionary, they are arranged in the order of descending frequencies. Then we replace words in the text with their indices in the dictionary. The text consists of 37,003,242 words and has the entropy H0 52,805 KB.
- The second scheme was proposed by Ferragina and Venturini [9] to compress a sequence of n characters to its high-order entropy so that a $O(\log n)$-bit substring can be decoded in constant time. The text is split into blocks of $\frac{1}{2}\log n$ bits, which are sorted by frequency and encoded as in the first scheme. In our test $n = 209,715,202$, which implies $\frac{1}{2}\log n \approx 14$. To reduce the volume of bit-level operations, we rounded the block size to 2 bytes. Since the alphabet is constructed of pairs of characters, the compressed text size should be compared with the entropy H1, which is 106,754 KB.

We measured the element extraction time and the space occupied by the encoded text together with auxiliary structures. The time was averaged over 100 million extractions of a random integer sequence element. To reduce the number of divisions in Algorithm 1, we chose the size both of L1 and L2-blocks as powers of 2: $L_1 = 2^{l_1}$ and $L_2 = 2^{l_2}$. The optimal chunk size in Algorithm 2 was determined experimentally and equals 7 bits for all tests.

Parameters l_1 and l_2 constitute a space/time trade-off shown in Fig. 4. Parameter l_2 has more impact because it defines the average number of iterations of the loops in Algorithm 1 and the size of arrays Δ_c and Δ_b. As mentioned above, the most prominent values of l_2 for our data are between 6 and 8. When l_2 decreases, arrays Δ_c and Δ_b become bigger, but loops have fewer iterations. This speeds up the algorithm by the cost of space until arrays become too big to fit into the L2 or L3 cache, which causes many cache mismatches. The latter situation is illustrated in Fig. 4b, where the element extraction for $l_2 = 6$ executes longer and requires more space than for $l_2 = 7$.

Two competitive solutions discussed in the Introduction were tested for the comparison: the Directly Addressable variable-length Codes (DAC) [6] and the Simple

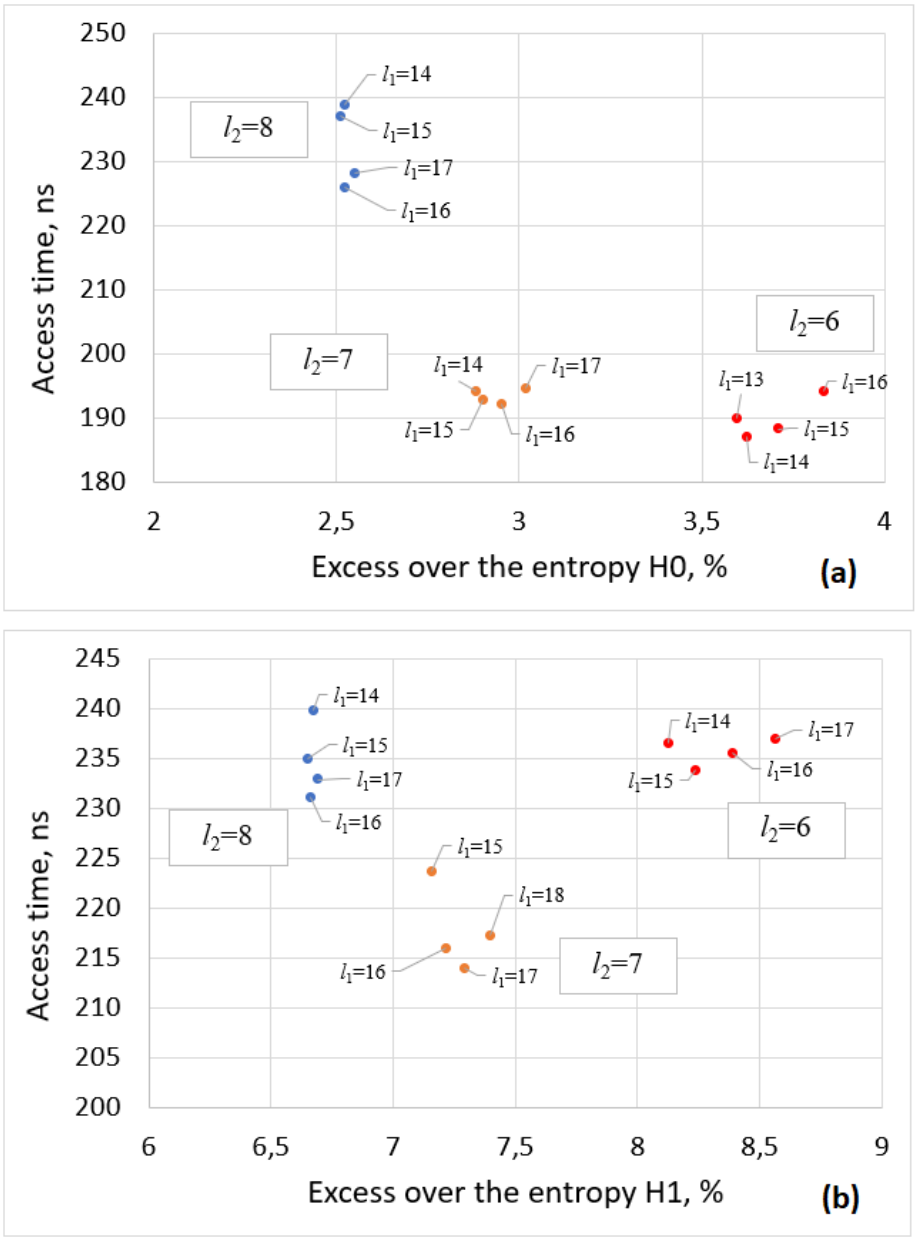

Figure 4. Element extraction from the RMD-encoded integer sequence: (a) word-based alphabet, (b) character-based alphabet

Dense Coding (SDC) [10]. The DAC relies on a 'black-box' rank operation for a bit-sequence, while random access via the SDC structure requires a select. The comparison of the best recent approaches to computing rank and select for binary sequences is given in [13] and [15]. In both sources, the two fastest methods to compute rank appear to be the Rank9-V1 [18] and its variation, the so-called IL (interleaving) [11], where the original bit-vector is interleaved with rank data. They also require relatively small space overhead (usually the Rank9-V1 uses somewhat bigger space than the IL).

As reported in [13], very fast 'select' algorithms operating at approximately the same speed are provided with SD [14], MCL, RSAA [13], and LA [5] structures. However, the space complexity highly depends on the percentage of ones in a bit-

vector. In the first scheme of our test set, it is about 13% and about 19% in the second scheme. For bit-vectors with a low percentage of ones, SD and LA select structures occupy more attractive positions on the space/time plane than the other two mentioned solutions. To implement them, we stored the simple dense code of the integer sequence as-is, while the auxiliary binary sequence needed for constant time random access is given in the form of a compressed LA- or SD-vector.

Also, we tested the naive method mentioned as "approach 1" in Introduction. The integer sequences were encoded with the Elias δ-code, and the position of each s-th codeword was sampled. To retrieve an element, we perform a sequential search from the sampled position.

The compressed file sizes, together with auxiliary data structures as well as average integer extraction times, are shown in Table 1 and in Fig. 5 (except for the naive approach in the Figure as it goes beyond the scale). The excess over the H0 entropy for the word alphabet and over the H1 entropy for the character-based one is shown in percentage. All data is stored in RAM. To build our and other solutions, we used the g++ compiler v9.4.0 with the -O3 optimization flag. We got IL and SD implementations from the SDSL library [16] and LA implementation from [4]. Tests have been run on a computer with an AMD Athlon 3000G processor, 32 KB of L1 cache, 512 KB of L2 cache, 4 MB of L3 cache, 16 GB RAM, and OS Ubuntu 20.04 LTS. The source code can be downloaded from [21].

We tested methods with different parameters representing different points in the space/time trade-off. For the first scheme, the code $R_{2,4-\infty}$ gives the best compression ratio, while for the second scheme, it is $R_{2-\infty}$. In each case, we show two pairs of parameters l_1, l_2 giving the best time and the best space, as well as space or time optimal LA parameters for the SDC+LA scheme. The DAC code is parameterized by the bit size of a chunk, b. The extraction time for the Elias δ-code depends on the sampling interval s.

As seen, the Elias codes are obviously space inefficient. Even the code itself exceeds the entropy H0 by 34% for the word-based alphabet and by 38% for the character-based. However, the extraction time can be quite low if the sampling rate is high. In fact, this way, we approach the uncompressed integer sequence.

Our data structure based on RMD codes is significantly more compact than all competitive solutions both for word-based and character-based alphabets. Also, our direct access method is faster than SDC in combination with the space-optimal LA or SD on both alphabets. However, the SDC+LA scheme may become faster at the cost of extra space ($bpc = 7$).

We tested DAC with different bit sizes of a chunk: $b = 4$ or $b = 8$. This parameter represents a space/time trade-off: operating whole bytes ($b = 8$) is much faster but requires much more space than for $b = 4$. The value $b = 2$ appears to be inefficient and is not shown in the tables.

Our method is better than DAC-4 in space and time on the word-based alphabet: $3-10\%$ shorter and $1-25\%$ faster, depending on parameters l_1, l_2 and DAC variations. Enlarging the chunk size to 8 bits makes DAC $2-2.7$ times faster than RMD by the cost of space (it becomes $15-19\%$ larger).

On the character-based alphabet, the encoded text becomes larger, and the size of the arrays $L1Byte$, $L2Length$, Δ_c, and Δ_b also grows, causing more cache mismatches and slowing down our algorithm. At the same time, encoded integers are smaller, requiring less number of streams in DAC. As a result, on the character-based alphabet, our method becomes slower even than DAC-4. However, its space outperformance

Integer sequence generated from the word-based alphabet

Algorithm	Parameters	Size, KB	Time, ns
Elias δ	$s = 4$	107,835 (104.9%)	127
	$s = 512$	71,121 (34.69%)	1999
RMD, $R_{2,4-\infty}$	$l_1 = 14$, $l_2 = 6$	54,719 (3.62%)	187
	$l_1 = 16$, $l_2 = 8$	54,138 (2.52%)	226
DAC	$b = 8$, IL	63,163 (19.61%)	86
	$b = 8$, Rank9-v1	64,387 (21.93%)	85
	$b = 4$, IL	57,595 (9.07%)	233
	$b = 4$, Rank9-v1	59,628 (12.92%)	228
SDC+LA	$bpc = 7$	67,634 (28.08%)	137
	v-opt	65,500 (24.04%)	253
SDC+SD		64,164 (21.51%)	300

Integer sequence generated from the character-based alphabet

Elias δ	$s = 4$	252,732 (136.7%)	135
	$s = 512$	148,694 (39.3%)	1932
RMD, $R_{2-\infty}$	$l_1 = 17$, $l_2 = 7$	114,538 (7.29%)	214
	$l_1 = 16$, $l_2 = 8$	113,863 (6.65%)	231
DAC	$b = 8$, IL	136,444 (27.81%)	47
	$b = 8$, Rank9-v1	135,935 (27.33%)	53
	$b = 4$, IL	124,133 (16.28%)	175
	$b = 4$, Rank9-v1	129,734 (21.5%)	150
SDC+LA	$bpc = 7$	159,914 (49.8%)	146
	v-opt	143,696 (34.61%)	323
SDC+SD		143,301 (34.23%)	433

Table 1. Experiments on integer compression and extraction

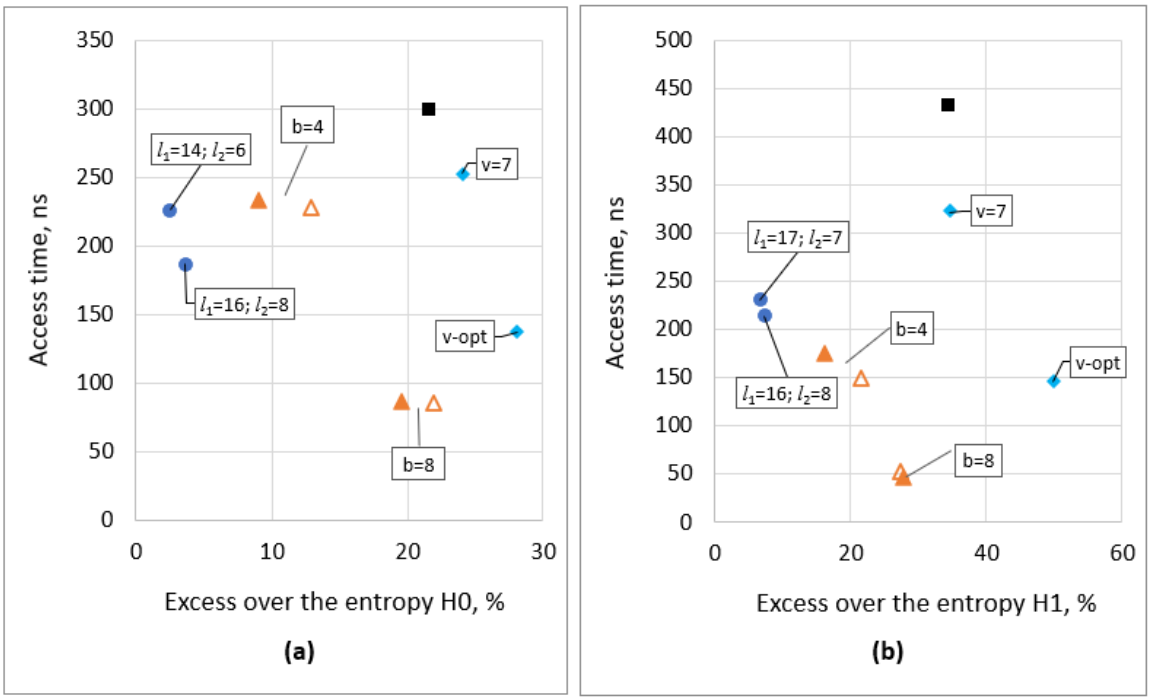

Figure 5. Experiments with different approaches: (a) word-based alphabet, (b) character-based alphabet

increases (the space optimal RMD structure is shorter than DAC by $9 - 14\%$ for $b = 4$ and $19 - 20\%$ for $b = 8$).

6 Conclusion

We presented a fast method of extracting an element of an unordered integer sequence compressed using the Reverse Multi-Delimiter codes. By exploiting the recently developed technique of linear approximation of a codeword block position and properties of the RMD codes, we achieved a very good compression ratio for integer sequences taken from a frequency-based compression of the 200MB English text. Together with all data structures required for fast direct access, the size of the compressed file exceeds the zero-order entropy on the word-based alphabet by $2.5 - 3.5\%$ and the first-order entropy on the character-based alphabet by $6.5 - 7.5\%$. At the same time, our method provides a decent speed of element extraction, being the fastest among competitive solutions that compress the text with a ratio exceeding the entropy by less than 15%.

References

1. A. Anisimov and I. Zavadskyi: *Variable-length prefix codes with multiple delimiters.* IEEE Transactions Information Theory, 63(5) 2017, pp. 2885–2895.
2. A. Anisimov, I. Zavadskyi, and T. Chudakov: *Practical word-based text compression using the reverse multi-delimiter codes*, in Information Technology and Implementation (ITI-2022), CEUR Workshop Proc., 2022, pp. 175—183.
3. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations.* IEEE Transactions Information Theory, 33 1987, pp. 238–245.
4. A. Boffa, P. Ferragina, and G. Vinciguerra: *Learned-compressed-rank-select.* https://github.com/aboffa/Learned-Compressed-Rank-Select-TALG22.
5. A. Boffa, P. Ferragina, and G. Vinciguerra: *A learned approach to design compressed rank/select data structures.* ACM Transactions on Algorithms, 2022.
6. N. R. Brisaboa, S. Ladra, and G. Navarro: *Directly addressable variable-length codes*, in String Processing and Information Retrieval, J. Karlgren, J. Tarhio, and H. Hyyrö, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 122–130.
7. D. R. Clark: *Compact Pat Trees*, PhD thesis, University of Waterloo, 1996.
8. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Transactions Information Theory, 21 1975, pp. 194–203.
9. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*, in Proc. 18th SODA, 2007, pp. 690—696.
10. K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in International Workshop on Experimental and Efficient Algorithms, 2007, pp. 203—216.
11. S. Gog and M. Petri: *Optimized succinct data structures for massive data.* Software: Practice and Experience, 44 2014, pp. 1287 – 1314.
12. G. Jacobson: *Succinct Static Data Structures. Ph.D. Thesis*, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.
13. O. Kulekci: *Counting with prediction: Rank and select queries with adjusted anchoring*, in 2022 Data Compression Conference, 2022, pp. 409–418.
14. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, Louisiana, USA, 2007, Society for Industrial and Applied Mathematics, pp. 60–70.
15. G. Pibiri and S. Kanda: *Rank/select queries over mutable bitmaps.* Information Systems, 99 2021, p. 101756.
16. *Succinct data structure library*: https://github.com/simongog/sdsl/.
17. L. Thiel and H. Heaps: *Program design for retrospective searches on large data bases.* Information Storage and Retrieval, 8(1) 1972, p. 1–20.
18. S. Vigna: *Broadword implementation of rank/select queries*, in Proceedings of the International Workshop on Experimental and Efficient Algorithms, 2008, pp. 154—-168.

19. I. Zavadskyi and A. Anisimov: *Reverse multi-delimiter compression codes*, in 2020 Data Compression Conference, 2020, pp. 173–182.
20. I. Zavadskyi and V. Zavadska: *Reverse multi-delimiter codes in English and Ukrainian natural language text compression*, in CEUR Workshop Proc., 2022, pp. 211—219.
21. I. O. Zavadskyi: *Direct access to RMD-encoded sequence elements. Implementation in C programming language*. https://github.com/zavadsky/DirectAccess.