Prefix Coding Scheme Supporting Direct Access without Auxiliary Space

Na Wang, *Member, IEEE,* Wei Yan, *Member, IEEE,* Hao Jiang, Sian-Jheng Lin, *Member, IEEE,* and Yunghsiang S. Han, *Fellow, IEEE*

Abstract—Entropy coding is a widely used technique for lossless data compression. The entropy coding schemes supporting the direct access capability on the encoded stream have been investigated in recent years. However, all prior schemes require auxiliary space to support the direct access ability. This paper proposes a rearranging method for prefix codes to support a certain level of direct access to the encoded stream without requiring additional data space. Then, an efficient decoding algorithm is proposed based on lookup tables. The simulation results show that when the encoded stream does not allow additional space, the number of bits per access read of the proposed method is above two orders of magnitude less than the conventional method. In contrast, the alternative solution consumes at least one more bit per symbol on average than the proposed method to support direct access. This indicates that the proposed scheme can achieve a good trade-off between space usage and access performance. In addition, if a small amount of additional storage space is allowed (it is approximately 0.057% in the simulation), the number of bits per access read in our proposal can be significantly reduced by 90%.

Index Terms—Data compaction and compression, Entropy Coding, Prefix Codes, Direct Access, and Additional Data Space.

1 Introduction

With the explosive growth of data volume, data compression has become an indispensable part of the storage and transmission of text [1], [2], images, sensor data [3] and trajectory data [4], etc. Prefix coding [5], [6] exerts a major role in well-known compression schemes. It is a type of variable-length coding [7] that the coding system does not have a codeword that is a prefix of any other codeword. Several well-known prefix codes include Huffman coding [8], [9], [10] and universal coding [11], [12], [13]. These codes often eliminate redundancy and improve transmission, storage and processing efficiency.

To reduce the space of a column-oriented database, each column can be compressed via a prefix code, such as Huffman coding. In this case, when a user wants to access the i-th entry of the column, all preceding entries shall be decoded, as the location of the i-th entry is unknown on the encoded stream. This paper refers to this method as the conventional method. Hence, the coding schemes that can decode an entry in an encoded stream have been investigated in recent years [14], [15], [16], [17]. A direct solution to this problem is to maintain an additional array, which stores the locations of certain entries [18], [19] to help to decode. For example, we can use an array to store the locations of each n-th entry, $n = 100, 200, ...$, on the encoded stream. However, using auxiliary storage space degrades the compression ratio of the coding. Zhou et al. [20] introduced a neighbor-based block-organized storage scheme by paying a small price for storage space to support local retrieval. However, it relies on the length-prefix compression algorithm (to be defined precisely in Section 2.2.3). In [21], Brisaboa et al. introduced Directly Addressable Codes (DACs), which present a variable-length encoding scheme for sequences of integers and enable direct access to any element of the source sequence. However, this method is only suitable for Vbyte encoding [22], and it requires additional space to achieve *rank* search in a constant time. Therefore, it is not a general solution to provide direct access capability in a variable-length encoded bit sequence. In [23], a new application of wavelet tree is introduced to provide accessibility and unique decodability for non-prefix-free codes. Nevertheless, storing the entire wavelet tree requires storage space, which brings a new overhead space. The problem of source coding with random access has also received attention from the information theory community [24], [25], [26], [27], [28], [29], [30]. Particularly, Mazumdar et al. [26] gave a fixed-length compression scheme that requires $\Theta(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ bits on the encoded stream to decode a single source bit, where $\varepsilon > 0$ is the rate exceeding the source entropy. Building on the work of [26], Tatwawadi et al. [27] discussed a systematic scheme to achieve close to optimal compression with finite random access. However, the researches in [26] and [27] only investigate random access to a single bit. In [29], an explicit entropy-achieving scheme that achieves constant average local decodability and update efficiency is given. Nevertheless, the probability of its local decoding may be

- *Na Wang is with the School of Optoelectronic Information and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China. E-mail: wna@usst.edu.cn.*
- *Wei Yan is with the Theory Laboratory, Huawei Technologies Company Ltd., Hefei 230000, China. E-mail: yanwei87@huawei.com.*
- *Hao Jiang is with Alibaba Group, Hangzhou 311121, China. E-mail: curry30@mail.ustc.edu.cn.*
- *Sian-Jheng Lin is with the Theory Laboratory, Huawei Technologies Company Ltd., Hong Kong, China. E-mail: lin.sian.jheng1@huawei.com.*
- *Yunghsiang S. Han is with the Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen 518110, China. E-mail: yunghsiangh@gmail.com.*

This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFA1004902, the National Natural Science Foundation of China under Grant 62071446, and the Open Fund of Shanghai Big Data Management System Engineering Research Center. (*Corresponding author: Sian-Jheng Lin.*)

nonzero and requires a particular data structure for codeword location. All existing methods require additional space to achieve direct access capability. That is, the compression ratios of existing methods are inferior to that of Huffman coding. In practice, the use of extra space is not suitable for all applications. For example, in wireless sensor networks [31], [32], energy consumption is a major bottleneck due to the limited memory storage, small battery capacity, limited processing power, and bandwidth of its tiny sensor nodes. Data compression technology has been used to reduce energy consumption and prolong the network's lifetime. It reduces data size before it is forwarded from sensor nodes to sink nodes in the network. A common operation in compressed data is direct access. However, in such a complex and constrained environment, the additional space overhead is generally unbearable for the tiny nodes. In contrast, a compression method that supports direct access without requiring additional space would be preferable.

In this paper, we extend [20] and propose a rearranging method for prefix codes to support direct access without requiring additional data space, which is reflected in the following two perspectives. First, the space overhead in the proposed method to store the encoded stream is equal to that in Huffman coding. Second, the cost for the proposed method to implement the direct access processing is independent of the volume of the input data but only related to the alphabet size. Further, in data transmission, due to the limited storage resources and network bandwidth, it is beneficial to reduce the storage space of the encoded data as much as possible. In contrast, the prior work [20] requires auxiliary space to mark the length of each codeword and the size of each block is fixed, resulting in a waste of some space. To the best of our knowledge, this paper is the first that employs the rearranging method for prefix codes and allows the average codeword length to be a floating-point number.

The contribution of this paper is summarized as follows.

1) A rearranging method for prefix codes is proposed to support direct access to the encoded stream without requiring auxiliary space.
2) A novel lookup table construction method and fast decoding algorithm are presented.
3) The simulations are given.
   a) If the encoded stream does not allow additional space: the number of bits per access read of the proposed method is above two orders of magnitude less than the conventional method; when the chosen prefix codes meet certain conditions, such as canonical Huffman code, the number of bits per access read can be further reduced.
   b) If the encoded stream allows using additional space: the alternative solution consumes on average at least one more bit per symbol than the proposed method to support direct access; when using a little additional space (in our simulation, it is approximately 0.057%), the number of bits per access read in the proposed method can be significantly reduced by 90%.

The remainder of the paper is organized as follows. Section 2 lists the notations and related works. Section 3 introduces the proposed scheme and algorithms. Section 4 presents the access algorithm for certain codes. Section 5 gives the simulation results and analysis, and Section 6 concludes this work.

2 Preliminaries

2.1 Notations

Let $\Sigma = \{z_1, z_2, \ldots, z_\sigma\}$ denote an alphabet, and Σ^* denote the set of all strings over Σ. Let $T = \{t_i\}_{i=1}^{N} = (t_1, t_2, \ldots, t_N)$ denote an N-symbol input sequence, for each $t_i \in \Sigma$. The i-th element of T is denoted as $T[i] = t_i$, and a subsequence of T is denoted by $T[i:j] = (t_i, t_{i+1}, \ldots, t_j)$.

The coding scheme $C : \Sigma \rightarrow W$ maps an alphabet Σ to a set of codewords $W = \{w_1, w_2, \ldots, w_\sigma\}$, where w_i is the codeword of z_i. Thus, the encoding of T generates a sequence $C(T) = (c_1, c_2, \ldots, c_N)$, where $c_i \in W$ is the codeword of $t_i \in \Sigma$. The encoded sequence has $|C(T)| = |c_1|+|c_2|+\cdots+|c_N|$ bits, where $|c_i|$ denotes the number of bits of c_i. In addition, we define an unconventional decoding function $\overline{C}(B)$ for a sequence B. Note that if the sequence B is decodable, $\overline{C}(B)$ returns two values: a decoded symbol and the reading length required to decode the symbol. Otherwise, $\overline{C}(B)$ returns $NULL$. In other words, $\overline{C}(B)$ is defined as:

$$\overline{C}(B) = \begin{cases} (t_i, r) & \text{if } \exists\, C(t_i) = B[1:r]; \\ NULL, & \text{otherwise;} \end{cases} \quad (1)$$

where $t_i \in \Sigma, 1 \le r \le |B|$. Therefore, if $\overline{C}(B) \neq NULL$, we conclude that the sequence B is decodable and the first r bits in B can decode a symbol t_i. Otherwise, we say B is not decodable. Table 1 lists the notations used throughout the paper for ease of reference.

2.2 Related works

2.2.1 Canonical Huffman code

Canonical Huffman code [33], [34] is a prefix code by remapping the codewords of Huffman coding. In the codebook of the canonical Huffman code, the symbols are sorted by their bit lengths in non-decreasing order. Figure 1 shows a Huffman tree and its corresponding canonical Huffman tree. In this way, in canonical Huffman code, the set of codewords of the same length can be seen as binary representations of consecutive integers. This property allows us to determine the codeword length by only reading a portion of the codeword. For example, in Figure 1(b), the length of the codeword is two if its prefix is 0 and is three if its prefix is 1.

2.2.2 Sampled Huffman code with R/S dictionaries [35]

Figure 2 depicts an example for an input sequence $T = NONPREFIXFREE$, where $Huffman(T)$ is the Huffman-encoded stream, and $A(Huffman(T))$ is the additional array. Precisely, $A(Huffman(T))$ is a $|C(T)|$-bit binary stream that marks the start locations of the $\{iS+1|i = 0, 1, ...\}$-th codewords in $Huffman(T)$. In Figure 2, when $S = 2$, the beginnings of $1, 3, 5, 7, 9, 11, 13$-th codewords are marked with 1, and the rest are marked with 0. Additionally, the underlined bits indicate the corresponding codewords at the sampling locations. To reduce space complexity, the additional array $A(Huffman(T))$ is compressed with an R/S

TABLE 1: Definitions of notations used throughout the paper

Symbol	Definition
Σ	An alphabet $\{z_1, z_2, \ldots, z_\sigma\}$
Σ^*	The set of all strings over Σ
$\{0,1\}^*$	$\{0,1\}^* = \{0, 1, 00, 01, 10, \cdots\}$ denotes the set of all binary strings
$\{t_i\}_{i=1}^N$	$(t_1, t_2, \ldots, t_N)$
T	An input source sequence
N	The number of symbols in the source sequence
$T[i]$	The i-th element of T
$T[i:j]$	$(T[i], T[i+1], \ldots, T[j])$
$\|B\|$	The length of $B \in \{0,1\}^*$
$C(\cdot)$	A coding scheme for a source sequence
$\overline{C}(\cdot)$	An unconventional decoding function. When the incoming sequence is decodable, it returns two values: a decoded symbol and the length required for decoding the symbol. Otherwise, it returns NULL
t	$t = \|C(T)\|/N \in \mathbb{R}$ is the average block size
$PopLast$	An index of the block, which stores the bit popped last from the stack during the rearranging
$sp(i)$	The start position of the i-th block
$bs(i)$	The size of the i-th block
L_i	The codeword length of the i-th symbol
p_i	The prefix that can determine L_i
L_{max}	The maximum codeword length in a codebook
B_i	The bit sequence stored in the i-th block
o_i	The overflow variable to indicate whether the i-th block overflows
$Cumu(i,j)$	The cumulative overflow variable of j consecutive blocks following the i-th block
$V_{alid}(C)$	The valid sequence available for decoding in a sequence C
$L_{valid}(C)$	The length of sequence $V_{alid}(C)$. Note that when C is all zeros, $L_{valid}(C) = 0$
$CountZeros(C)$	The number of consecutive zeros in the sequence C starting from the least significant bit
$Modify(C)$	Modify a sequence C
$LookUp(C)$	Uses sequence C for the table lookup, and returns two attributes: *sym* and *len*
$L(i)$	Starting block index when decoding the i-th symbol
$H(i)$	Ending block index when decoding the i-th symbol
$\langle L(i), H(i)\rangle$	An envelope structure
$\langle L(m_i), H(m_i)\rangle$	The maximum envelope of $\langle L(i), H(i)\rangle$

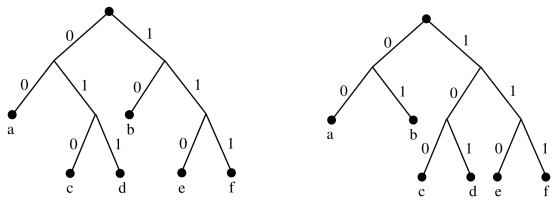


Fig. 1: (a). A Huffman tree, and (b). its corresponding canonical Huffman tree

dictionary data structure that allows constant time *select* queries. To access the i-th symbol, we simply query compressed $A(Huffman(T))$ with $j = select(\lceil \frac{i}{S} \rceil)$, and begin decoding from location j in $Huffman(T)$ until the i-th symbol is obtained.

2.2.3 Neighbor-based storage scheme

Given n independent sequences $\{T_i\}_{i=1}^n$, these sequences are compressed to $\{C(T_i)\}_{i=1}^n$ and stored in an array X. If these compressed sequences are stored in X with a constant interval $\bar{t}$, one can read each $C(T_i)$ from X directly. That is, each $C(T_i)$ is stored in $X[\bar{t} \times (i-1) + 1 : \bar{t} \times i]$. However, as

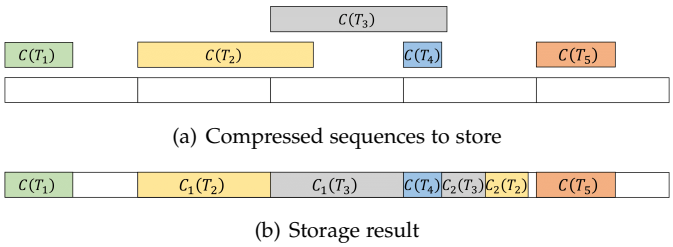
$T = NONPREFIXFREE$

Σ	E	R	F	N	I	O	P	X
W	01	00	100	101	1110	1111	1100	1101

$Huffman(T) = \underline{101}1111\underline{101}1100\underline{00}01\underline{100}1110\underline{1101}100\underline{00}0\underline{101}$
$A(Huffman(T)) = 1000000100000010001000000100000010000100$

Fig. 2: A sampling example in [35]

$\{|C(T_i)|\}_{i=1}^n$ are variant, if we choose $\bar{t} = \max_{i=1}^n |C(T_i)|$, this causes fragmentation and degrades the compression ratio.



(a) Compressed sequences to store

(b) Storage result

Fig. 3: Neighbor-based scheme: storage example

Zhou et al. [20] introduced a rearranging scheme by paying a small price for storage space (around 3% extra storage space) to solve this issue. The scheme stores $\{C(T_i) = g'(T_i)||g(T_i)\}_{i=1}^{n}$, precisely, $C(T_i)$ is the concatenation of $g'(T_i)$ and $g(T_i)$, where $g : \Sigma^* \to \{0,1\}^*$ is a variable-length lossless compression without prefix constraints, and $\{0,1\}^* = \{0,1,00,01,10,\cdots\}$ denotes the set of all binary strings, and $g'(T_i)$ is the binary representation of the length of $g(T_i)$. Therefore, one can read the first $|g'(T_i)|$ bits in its corresponding block to obtain the length of $g(T_i)$.

Next, a neighbor-based storage scheme is proposed, where the part of the sequence that exceeds the block size is called overflow. The store rules are given below.

1) If $|C(T_i)| \le \bar{t}$, then $X[\bar{t}\times(i-1)+1 : \bar{t}\times(i-1)+|C(T_i)|] \leftarrow C(T_i)$. A visual example is given in Figure 3(b), as compressed sequences $C(T_1)$, $C(T_4)$ and $C(T_5)$ shown.
2) If $|C(T_i)| > \bar{t}$, then $C(T_i) = C_1(T_i)||C_2(T_i)$ is divided into two parts such that $|C_1(T_i)| = \bar{t}$, and $X[\bar{t}\times(i-1)+1 : \bar{t}\times i] \leftarrow C_1(T_i)$. The overflow part $C_2(T_i)$ is stored in the neighboring blocks that have free space according to the nearest neighbor priority. For example, for the free space in block 4 in Figure 3(b), the overflow of $C(T_3)$ has storage priority, as $C_2(T_3)$ shown in Figure 3(b). If $C_2(T_3)$ has been stored, the rest space is used to store the overflow of $C(T_2)$, denoted by $C_2(T_2)$. Notably, to fill as many free spaces as possible, the blocks are designed as cyclic, i.e., the next block of the final block is the first block.

Next, the method to access T_i in X is presented below. First, we read the prefix $g'(T_i)$ of $C(T_i)$ to determine the length $|g(T_i)|$. If $|g(T_i)| \le \bar{t} - |g'(T_i)|$, i.e., $|C(T_i)| \le \bar{t}$, the next $|g(T_i)|$ bits are simply read to complete the retrieval. Otherwise, we read $g'(T_{i+1})$ in the $(i+1)$-th block to determine whether data needs to be read from the $(i+1)$-th block. This process is repeated until the decoding of T_i is finished. As shown in Figure 3(b), to access T_2, the prefix $g'(T_2)$ in the 2-th block is first read. Since $|C(T_2)| > \bar{t}$, it is known that 2-th block is overflow and the overflow variable is $o_2 = |C(T_2)| - \bar{t}$. Then, we repeat this process for the 3-th block and get the overflow variable $o_3 = |C(T_3)| - \bar{t}$. When we move to the 4-th block, we know that the 4-th block stores overflow data. Finally, we read the overflow of $C(T_2)$ starting at location $|C(T_4)| + o_3 + 1$ in the 4-th block to complete the decoding.

3 Proposed Rearranging Method and Algorithms

In this section, we propose a rearranging method that does not require the average block size to be an integer. Then, a lookup table for decoding is given. Finally, we present the encoding, decoding and direct access algorithms.

3.1 Rearranging method

Given a source sequence T of N symbols, the proposed method generates a $|C(T)|$-bit sequence X via a prefix code with the block sizes $\lfloor t \rfloor$ or $\lfloor t \rfloor + 1$ bits, where $t = |C(T)|/N \in \mathbb{R}$. Notably, t could be a floating-point number, and [20] is only applicable to the case that the block size is a fixed integer number such that the approach [20] cannot be applied directly in our method. In contrast, the proposed method allows each block to have $\lfloor t \rfloor$ or $\lfloor t \rfloor + 1$ bits. Precisely, the codeword of the i-th symbol is expected to be stored in $X[sp(i) : sp(i+1) - 1]$, where $sp(i)$ is the start position of the i-th block in the compressed sequence X, which can be calculated by

$$sp(i) = \lfloor (i-1) \times t \rfloor + 1. \tag{2}$$

It can be seen that the i-th block can store $bs(i) = sp(i+1) - sp(i)$ bits.

The strategy for storing $C(T_i)$ in $X[sp(i) : sp(i+1) - 1]$ is similar to Section 2.2.3. That is, when the length of $C(T_i)$ exceeds the size of the i-th block, i.e., $|C(T_i)| > bs(i)$, we store the overflow part of size $|C(T_i)| - bs(i)$ in the neighboring blocks that have free space according to the nearest neighbor priority. Otherwise, let $X[sp(i) : sp(i) + |C(T_i)| - 1] \leftarrow C(T_i)$. Finally, we can obtain a rearranged bit sequence.

Once the rearranged sequence is obtained, the procedure to decode the i-th symbol is as follows. First, we read the sequence stored in the i-th block, denoted by a key K. According to the above description, there are three cases for decoding the i-th symbol. In the first, if K is part of the codeword of the i-th symbol, we need to find its overflow from its subsequent blocks to complete the decoding. Concretely, we temporarily store K into a stack and try to find overflow from the next block. If the next block does not store the overflow, then we continue with the block following it. Note that the overflow may be divided into several parts and stored in multiple blocks according to the store rules. Therefore, whenever we read overflow in a certain block, we pop K from the stack and concatenate it with the overflow read, getting a new sequence $\hat{K}$, and try to decode $\hat{K}$. If $\hat{K}$ is still not decodable, we push $\hat{K}$ into the stack again and continue the process until the i-th symbol is decoded. In the second, if K is just the codeword of the i-th symbol, we can directly decode K and output the i-th symbol. Finally, if K consists of the codeword of the i-th symbol followed by the overflow of several codewords preceding the i-th symbol, we can also output the i-th symbol directly. The unused bits in decoding (the unused bits refer to the overflow of some codewords preceding it) can be used to help decode the symbols before the i-th symbol. In this paper, the decoding is implemented based on a proposed novel lookup table (see Section 3.2 for details). Intuitively, we use the key K to look up the table to get the information needed, which can facilitate the decoding process.

3.2 Construction of decoding table for truncated codewords

In decoding prefix code, the method based on a lookup table is widely used [36]. From Section 3.1, it can be seen that a codeword $C(T_i)$ in the proposed method may be dispersed in multiple blocks. As the tail of the codeword is stored in later blocks, there is an issue that the codeword used for decoding is truncated. That is, we may not have enough information to decode the symbol. Thus the conventional lookup table method is not suitable.

A modified version of the lookup table construction is proposed to solve this issue. Let L_{max} denote the maximum codeword length in the codebook. The proposed lookup table has 2^m entries, where $m > L_{max}$ and each entry is a

binary m-bit string. The value of the incoming m-bit binary string acts as an index to the lookup table. In the lookup table, each entry has two attributes, termed *sym* and *len*, which represent the symbol decoded by the entry and the number of bits required to decode the symbol, respectively.

For each m-bit entry C in the lookup table, the valid sequence for decoding is defined as

$$V_{alid}(C) = C >> (CountZeros(C) + 1), \tag{3}$$

where >> denotes the bitwise right-discard operation, e.g., if $C = 0111$, then $C >> 2 = 01$, and $CountZeros(C)$ denotes the number of consecutive zeros in C starting from the least significant bit. For example, for a sequence $C = 10100$, we have $V_{alid}(C) = 10$. The rightmost 1 and its subsequent bits in C are essentially unavailable for decoding. The bits 100 in the sequence 10100 are not used for decoding. Next, the length of the sequence $V_{alid}(C)$ is given by

$$L_{valid}(C) = m - (CountZeros(C) + 1). \tag{4}$$

When C is all zeros, we define $L_{valid}(C) = 0$.

Let $B[1 : i]$ be the prior part of a sequence B, where $1 \le i \le |B|$. In the construction of the proposed lookup table, for an m-bit entry C, if the prior part of $V_{alid}(C)$ is identical to a codeword w_i in the codebook, where w_i is the codeword of symbol z_i, then we say C is decodable and have

$$sym \leftarrow z_i, \qquad len \leftarrow |w_i|. \tag{5}$$

Otherwise, C is undecodable. In this case, we define

$$sym \leftarrow \$, \qquad len \leftarrow 0, \tag{6}$$

where \$ is a symbol not in the alphabet. Therefore, the value of *len* can be used to determine whether C is decodable. Exactly, if $len > 0$, C is decodable; otherwise, it is not. From the above description, it can be seen that the cost of building the proposed lookup table is related to the selection of m and is independent of the volume of input data.

From (3), we can see that in our proposed lookup table, the rightmost one and its subsequent zeros in each entry are not available for decoding. Therefore, to support direct access to the lookup table, we need to modify the key K accordingly before the table lookup. Precisely, we use

$$K' \leftarrow Modify(K) = ((K \ll 1) + 1) \ll (m - |K| - 1) \tag{7}$$

to look up the table, where $\ll$ denotes the bitwise left-shift operation and sets zero at the least significant bit. Next, the operation

$$(sym, len) \leftarrow LookUp(K')$$

uses K' for the table lookup, and returns the corresponding attributes *sym* and *len*.

To illustrate the details of constructing the lookup table, an example is given below.

Example 1: Consider the codebook as

$$\begin{aligned} \Sigma &= (a, b, c), \\ W &= (0, 10, 11), \end{aligned} \tag{8}$$

and we have $L_{max} = 2$. For simplifying the representation, we choose $m = 3$ as an example. Next, the 2^3 entries in the lookup table are decoded. For the entry $C = 000$, as $L_{valid}(C) = 0$, thus we have $sym = \$$, $len = 0$. For the entry $C = 001$, $L_{valid}(C) = 3 - 1 = 2$, and the prior part of $V_{alid}(C)$ is identical to the codeword 0 in the codebook, which is the codeword of symbol a. Thus, the corresponding $sym = a$, $len = 1$. We proceed similarly with entry $C = 010$ and so on. Finally, the lookup table shown in Table 2 is obtained.

TABLE 2: The constructed lookup table for Example 1

C	*sym*	*len*
000	\$	0
001	a	1
010	a	1
011	a	1
100	\$	0
101	b	2
110	\$	0
111	c	2

Algorithm 1: Proposed rearranging algorithm

Input: A source sequence T of N symbols, the average block size t.
Output: An encoded sequence X.

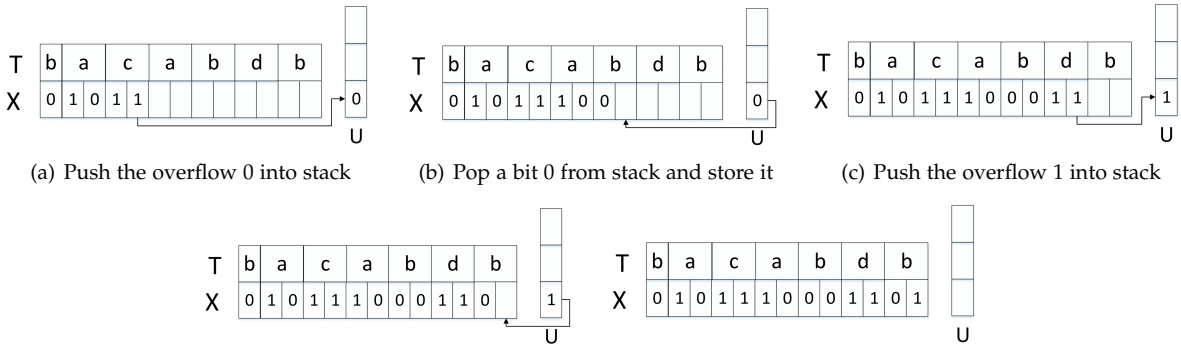
```
Allocate N × t bits to X;
Build a stack U;
for i = 1 to N do
    Calculate sp(i) and sp(i+1) via (2);
    if |C(T_i)| ≤ bs(i) then
        X[sp(i) : sp(i) + |C(T_i)| − 1] ← C(T_i);
        for i = sp(i) + |C(T_i)| to sp(i+1) − 1 do
            if U.size() ≠ 0 then
                X[i] ← U.pop();
    else
        Divide C(T_i) into C_1(T_i) and C_2(T_i);
        X[sp(i) : sp(i+1) − 1] ← C_1(T_i);
        U.push(C_2(T_i));
while U.size() ≠ 0 do
    for i = 1 to N × t do
        if X[i] is empty then
            X[i] ← U.pop();
```

3.3 Algorithms

In this subsection, we first present a rearranging algorithm for the proposed method by using a stack, where each entry in the stack is a bit. Then, the algorithm for decoding the source sequence is given. Finally, we provide an access algorithm based on the proposed lookup table.

3.3.1 Rearranging

We discuss it in the following three cases. In the first, if $|C(T_i)| > bs(i)$, we divide $C(T_i)$ into two parts $C(T_i) = C_1(T_i)||C_2(T_i)$, where $|C_1(T_i)| = bs(i)$, and let $X[sp(i) : sp(i+1) - 1] \leftarrow C_1(T_i)$. Besides, we push $C_2(T_i)$ into the stack bit by bit in the rightmost first order. In the second, if $|C(T_i)| = bs(i)$, let $X[sp(i) : sp(i+1) - 1] \leftarrow C(T_i)$. Finally, if $|C(T_i)| < bs(i)$, let $X[sp(i) : sp(i) + |C(T_i)| - 1] \leftarrow C(T_i)$, and if the stack is not empty, we start popping up a bit from the stack and sequentially store it in $X[sp(i) + |C(T_i)| : sp(i+1) - 1]$, the popping process ends either if the stack is empty or $X[sp(i) + |C(T_i)| : sp(i+1) - 1]$ is full.



(a) Push the overflow 0 into stack (b) Pop a bit 0 from stack and store it (c) Push the overflow 1 into stack

(d) Pop a bit 1 from stack and store it (e) Rearranging finished

Fig. 4: A rearranging example

Algorithm 1 presents the proposed rearranging procedure. In Algorithm 1, Lines 5–9 handle the case $|C(T_i)| \leq bs(i)$. Lines 10–13 handle the opposite. Lines 14–17 handle the case that all symbols are encoded, but there is still overflow data in the stack. As the blocks are cyclic, we start from the first block looking for the free space in X and storing the data there.

To facilitate understanding, Figure 4 gives an example for encoding an input sequence $T = bacabdb$ with 7 symbols, and the corresponding alphabets and codewords are given by

$$\begin{aligned} \Sigma &= (a, b, c, d)\,, \\ W &= (10, 0, 110, 111)\,. \end{aligned} \tag{9}$$

The block sizes are either 1 or 2 in this example.

In addition, if one wants to recover the source sequence T with N symbols from the rearranged bit sequence (see below), we have to additionally store an index of the block, denoted by *PopLast*, where $1 \leq PopLast \leq N$, which stored the bit popped last from the stack during the rearranging, e.g., in Figure 4(d), since the 7-th block stores the bit 1 popped last from the stack, we have $PopLast = 7$. Notably, the block *PopLast* might have a small number of indexes since the blocks are located in cyclic, and storing it brings us a little additional overhead.

3.3.2 Decoding

The decoding algorithm for the source sequence is implemented using a stack, where each entry in the stack is also a bit. Let B_i denote the bit sequence stored in the i-th block. According to the description in Section 3.3.1, B_i can be divided into the following three cases. First, if B_i is a codeword, we can directly decode it to get the i-th symbol. In the second, if B_i is the prefix part of a codeword, we need to look for the overflow from subsequent blocks to complete its decoding. Finally, if B_i consists of a full codeword and the overflow of the blocks preceding it, we can decode it to get the i-th symbol, and the unused bits (i.e., the overflow of the blocks preceding it) can be used for decoding other symbols preceding it. In this subsection, to recover the entire source sequence T with N symbols, we start decoding from the *PopLast*-th block in the backward direction, where the index *PopLast* has been recorded during the rearranging process (see Section 3.3.1). There are N steps in the backward decoding. Precisely, let

$$\mu = \begin{cases} PopLast + 1 - i & \text{if } 1 \leq i \leq PopLast, \\ N + PopLast + 1 - i & \text{if } PopLast + 1 \leq i \leq N. \end{cases} \tag{10}$$

In step i, where $1 \leq i \leq N$, we read sequence B_μ in the μ-th block to decode $T[\mu]$. For example, in Figure 4, as $PopLast = 7$, we read B_7 in step 1 to decode $T[7]$, then we read B_6 in step 2 to decode $T[6]$, and so on. The reason for decoding in the backward direction is to first obtain the overflow required by the codewords to be decoded later to facilitate its decoding. Accordingly, if the read sequence is decodable, we decode it to get the corresponding symbol. We temporarily store unused bits on the stack if there are unused bits in the decoding. In subsequent steps, when we read a sequence that cannot be decoded, we achieve its decoding with the help of the bits in the stack (see below for details).

According to the store rules, in the backward decoding, B_μ in step i has the following three cases. In the first, if B_μ holds a full codeword and some overflow, i.e., $\overline{C}(B_\mu) = (T[\mu], r) \neq NULL$, where $1 \leq r < |B_\mu|$, we can directly obtain $T[\mu]$ and push the unused bits $B_\mu[r+1 : |B_\mu|]$ into stack bit by bit in rightmost first order. In the second, if B_μ is just a codeword, we decode it to get $T[\mu]$ and continue to the next step. Finally, if B_μ is a prefix part of a codeword, we start popping up a bit from the stack and sequentially append it to the end of B_μ, getting a new sequence $\hat{B_\mu}$, the popping process ends if $\overline{C}(\hat{B_\mu}) \neq NULL$. Notably, since we have stored the overflow required by the codewords to be decoded later in the previous steps, there are enough bits in the stack to help decode B_μ. When $\overline{C}(\hat{B_\mu}) \neq NULL$, we can decode it to obtain the μ-th symbol and continue to the next step until the entire source sequence is decoded.

Algorithm 2 presents the proposed decoding algorithm. In Algorithm 2, Lines 6–9 handle the case where B_μ can be directly decoded. In this case, we directly obtain $T[\mu]$, and if there are unused bits, we temporarily store them in the stack. Lines 10–13 handle the opposite. Particularly, Lines 11–12 use the unused bits in the stack to help decode it. Finally, Line 13 returns the decoded symbol $T[\mu]$. An example is given below to illustrate Algorithm 2.

Example 2: Given a bit sequence $X = 0101110001101$ with 13 bits, $N = 7$ and $PopLast = 7$. The codebook is given in

Algorithm 2: Proposed decoding algorithm

Input: An encoded sequence X, the number of symbols N, the average block size t, the block index *PopLast*.
Output: A source sequence T.

Build an array T of length N, a stack U;
for $i = 1$ *to* N **do**
  Calculate μ via (10);
  Calculate $sp(\mu)$ via (2);
  $B_\mu \leftarrow X[sp(\mu) : sp(\mu+1) - 1]$;
  if $\overline{C}(B_\mu) \neq NULL$ **then**
    $(T[\mu], r) \leftarrow \overline{C}(B_\mu)$;
    if $r < |B_\mu|$ **then**
      push $B_\mu[r+1 : |B_\mu|]$ into stack U bit by bit in rightmost first order;
  else
    while $\overline{C}(B_\mu) = NULL$ **do**
      $B_\mu \leftarrow B_\mu + U.pop()$;
    $(T[\mu], r) \leftarrow \overline{C}(B_\mu)$;

(9), and each block's start and end locations are determined by (2). The backward decoding procedure starting from the *PopLast*-th block is as follows. First, the sequence 01 in the 7-th block is read and decoded to a symbol b with an unused bit 1; then we have $T[7] = b$ and push bit 1 into the stack. Next, we move to the 6-th block and read the sequence 11. Since $\overline{C}(11) = NULL$, thus we pop bit 1 from the stack and append them to the end of the sequence 11, getting a new sequence 111. As $(d, 3) \leftarrow \overline{C}(111)$, then $T[6] = d$ is determined. We proceed similarly in subsequent steps until all symbols are decoded. Finally, we get the source sequence $T = bacabdb$.

Algorithm 3: $READ\_LOOKUP(i)$

Input: An encoded sequence X, a block index i, the proposed look-up table.
Output: (B'_i, sym_i, len_i), where B'_i is the modified sequence and (sym_i, len_i) is a pair of attributes after the table lookup.

$B_i \leftarrow X[sp(i) : sp(i+1) - 1]$;
$B'_i \leftarrow Modify(B_i)$;
$(sym_i, len_i) \leftarrow LookUp(B'_i)$;
Return (B'_i, sym_i, len_i);

3.3.3 Accessing

In this subsection, we show a method to access the i-th symbol via the lookup table with 2^m entries described in Section 3.2. The access algorithm is implemented based on two stacks, U_1 and U_2, where each entry in U_1 is with m bits and in U_2 with a bit.

To access the i-th symbol, we first read the sequence B_i in the i-th block. From Section 3.1 and Section 3.2, it can be seen that the length of B_i meets

$$|B_i| \in \{\lfloor t \rfloor, \lfloor t \rfloor + 1\} \leq L_{max} < m. \tag{11}$$

Algorithm 4: Proposed access algorithm with a lookup table

Input: An encoded sequence X, the number N of symbols, and an index i.
Output: The i-th symbol s.

Build two stacks U_1 and U_2;
$(C, s, l) \leftarrow READ\_LOOKUP(i)$;
if $l > 0$ **then**
  return s;
else
  push C into stack U_1;
  while $U_1.size() \neq 0$ **do**
    $k \leftarrow$ the index of the next block;
    $(C, s, l) \leftarrow READ\_LOOKUP(k)$;
    if $l = 0$ **then**
      push C into stack U_1;
    else
      if $l < L_{valid}(C)$ **then**
        push $C[l+1 : L_{valid}(C)]$ into stack U_2 bit by bit in rightmost first order;
        while $U_2.size() \neq 0$ **do**
          $E_t \leftarrow U_1.pop()$;
          $E_t \leftarrow V_{alid}(E_t)$;
          $l \leftarrow 0$;
          while $U_2.size() \neq 0$ *and* $l = 0$ **do**
            $E_t \leftarrow E_t + U_2.pop()$;
            $E'_t \leftarrow Modify(E_t)$;
            $(s, l) \leftarrow LookUp(E'_t)$;
        if $U_2.size() = 0$ *and* $l = 0$ **then**
          push E'_t into stack U_1;

Therefore, to facilitate direct table lookup, we first need to modify B_i through $B'_i \leftarrow Modify(B_i)$ to obtain an m-bit key B'_i. Next, we perform $(sym_i, len_i) \leftarrow LookUp(B'_i)$, i.e., using B'_i to look up the table. For clarity, Algorithm 3 defines a function $READ\_LOOKUP(i)$ that covers the above three steps: read B_i, modify B_i and look up the table using B'_i. Finally, $READ\_LOOKUP(i)$ returns the modified sequence B'_i and a pair of attributes (sym_i, len_i) after the table lookup. From (5)–(6), len_i has the following two cases.

1. If $len_i > 0$, B_i is decodable, thus the symbol sym_i is the desired output.
2. If $len_i = 0$, B_i is not decodable, we need to find its overflow from subsequent blocks to complete its decoding.

Next, we give the decoding details for Case 2 above. First, we push m-bit B'_i into stack U_1, then we move to the next block, proceeding with the procedure in Algorithm 3. For any subsequent block k, the table-lookup result $(B'_k, sym_k, len_k) \leftarrow READ\_LOOKUP(k)$ can be divided into the following three cases. Note that, from (4), the valid sequence length for decoding in B'_k is $L_{valid}(B'_k)$.

(i) If $len_k = 0$, B_k cannot be decoded. In this case, we push B'_k into stack U_1.
(ii) If $0 < len_k = L_{valid}(B'_k)$, B_k is just a codeword, without any overflow of other codewords. In this case, we move to the next block.

(iii) If $0 < len_k < L_{valid}(B'_k)$, B_k consists of a full codeword and some overflow of the blocks preceding it. In this case, if stack U_1 is not empty, we will use the unused bits (i.e., the sequence $B'_k[len_k + 1 : L_{valid}(B'_k)]$) to help decode the sequences in stack U_1.

We continue the above procedures until we read a block j that meets Case (iii). That is, we have $(B'_j, sym_j, len_j) \leftarrow READ\_LOOKUP(j)$ and $0 < len_j < L_{valid}(B'_j)$. Next, we use the unused bits $B'_j[len_j + 1 : L_{valid}(B'_j)]$ to help decode sequences stored in stack U_1. Concretely, we first push $B'_j[len_j + 1 : L_{valid}(B'_j)]$ into stack U_2 bit by bit in rightmost first order. Then, we pop m-bit sequence E_t from stack U_1. Note that the sequences stored in stack U_1 have been modified to m-bit, i.e., the valid sequence for decoding in any E_t is $V_{alid}(E_t)$. Next, we start popping up bit from stack U_2 and sequentially append it to the end of $V_{alid}(E_t)$, getting a new sequence $\hat{V_{alid}}(E_t)$, and then modify $\hat{V_{alid}}(E_t)$ to an m-bit E'_t to lookup the table. The popping process ends if the stack U_2 is empty or E'_t is decodable. When the popping ends, there are three cases (except for the case where stack U_2 is not empty and E'_t is not decodable, because in this case, we continue popping up bits from stack U_2 until stack U_2 is empty or E'_t is decodable) as follows.

1) If stack U_2 is empty, but E'_t is still not decodable, according to the store rules, the block storing E_t still has an overflow in the blocks following the j-th block. Therefore, we push E'_t into stack U_1 and move to the next block to find the overflow.
2) If stack U_2 is empty and E'_t is decodable, we have the following two cases. In the first, if stack U_1 is not empty (this indicates that the i-th symbol has not been decoded), we move to the next block and look for the overflow to help decode the sequences in stack U_1. In the second, if stack U_1 is empty, decoding the i-th symbol is achieved.
3) If stack U_2 is not empty and E'_t is decodable, we have the following two cases. First, if stack U_1 is not empty, we pop the m-bit sequence from stack U_1 and use the remaining bits in stack U_2 to help decode it. In the second, if stack U_1 is empty, we can conclude that decoding the i-th symbol is achieved.

We continue the above procedures until stack U_1 is empty. When stack U_1 is empty, decoding the i-th symbol is finished.

Algorithm 4 gives the corresponding algorithm details. In Algorithm 4, Lines 3–4 handle the case that the sequence in the i-th block can be directly decoded. Lines 5–24 handle the opposite. In particular, Lines 13–24 use the unused bits to help decode the sequences in stack U_1. Below we give an example to explain Algorithm 4.

Example 3: Consider a bit sequence $X = 110001$, $N = 4$ and a codebook shown in (8). The corresponding lookup table is given in Table 2. From (2), it can be determined that the block sizes are 1, 2, 1, 2, respectively. To access the first symbol, we have $B_1 = 1$, $C = Modify(B_1) = 110$. As $(\$, 0) \leftarrow LookUp(110)$ in Table 2, we push sequence 110 into stack U_1 and move to the next block. Next, we have $B_2 = 10$, $C = Modify(B_2) = 101$. Considering $(b, 2) \leftarrow LookUp(101)$ and $L_{valid}(101) = 2$, i.e., B_2 is just a codeword without any overflow of other codewords. Thus we move to the third block and have $B_3 = 0$, $C = Modify(B_3) = 010$. Similarly, B_3 is just a codeword. Next, we move to the 4-th block and have $B_4 = 01$, $C = Modify(B_4) = 011$. As $(a, 1) \leftarrow LookUp(011)$ and $L_{valid}(011) = 2 > 1$, we push the unused bit 1 in B_4 into stack U_2 and use it to help decode the top entry $E_t = 110$ in stack U_1. The valid sequence in E_t is $V_{alid}(E_t) = 1$, then we pop up bit 1 from stack U_2 and append it to the end of $V_{alid}(E_t)$, resulting in a new sequence $\hat{V_{alid}}(E_t) = 11$. Next, we have $C = Modify(\hat{V_{alid}}(E_t)) = 111$ and use sequence 111 to lookup the table. Since $(c, 2) \leftarrow LookUp(111)$ and stack U_1 are empty now, we can conclude that the first symbol in the source sequence is c.

4 Access algorithm with fewer bits for certain codes

Section 2.2.1 states that canonical Huffman code can determine the codeword length by its prefix. In this paper, a code whose codeword length can be determined by the prefix is called a canonical form code. In this section, we show that when the encoding scheme is performed on a canonical form code in the proposed rearranging method, the number of bits reads for direct access can be reduced.

Let L_i denote the codeword length of the i-th symbol, where $1 \le i \le N$. Let p_i denote the prefix that can determine L_i. Also, let $o_i = bs(i) - L_i$ be an overflow variable to indicate whether the i-th block overflows. According to the store rules, o_i has the following three cases. First, if $o_i = 0$, i.e., $bs(i) = L_i$, we can conclude that the sequence B_i in the i-th block is just a full codeword. In the second, if $o_i > 0$, i.e., $bs(i) > L_i$, we can conclude that B_i consists of a full codeword and the overflow of some blocks preceding it. In the third, if $o_i < 0$, i.e., $bs(i) < L_i$, we can conclude that B_i is a prefix part of a codeword, and its overflow is stored in subsequent blocks. Especially this section considers the case where the length of each prefix that can determine the codeword length does not exceed the corresponding block size, i.e., $|p_i| \le bs(i)$ for $1 \le i \le N$, where $|p_i|$ is the length of p_i. This can ensure that in any block i, we can read part of B_i or the full B_i to determine the codeword length L_i.

Before formally giving the algorithm details for Case $bs(i) > L_i$, two facts need to be highlighted. First, in finding the overflow of the i-th block, when we move to a subsequent block, we do not need to decode the symbol in the current block but only need to determine whether the current block stores the overflow of the i-th block. If it does, we read the overflow to help decode the i-th symbol. Otherwise, we should continue with the next block. Second, if any block k following the i-th block has free space after storing the codeword of the k-th symbol, it will first store the overflow of the nearest block preceding it. If a block occurs an overflow between the i-th block and the k-th block, the free space in the k-th block will store the overflow of this block first instead of the overflow of the i-th block.

The algorithm details are now given below. Let

$$Cumu(\alpha, \beta) = \sum_{k=\alpha+1}^{\alpha+\beta} o_k = \sum_{k=\alpha+1}^{\alpha+\beta} bs(k) - L_k = \sum_{k=\alpha+1}^{\alpha+\beta} bs(k) - \sum_{k=\alpha+1}^{\alpha+\beta} L_k$$

denote the cumulative overflow variable of β consecutive blocks following the α-th block, where $\alpha, \beta \in \mathbb{N}^+$. To find the

overflow of the i-th block, we first calculate $Cumu(i,1)$. If $Cumu(i,1) \leq 0$, i.e., $bs(i+1) \leq L_{i+1}$, the $(i+1)$-th block has no free space, thus it does not store the overflow of the i-th block. Then, we calculate $Cumu(i,2)$. If $Cumu(i,2) \leq 0$, i.e., $\sum_{k=i+1}^{i+2} bs(k) \leq \sum_{k=i+1}^{i+2} L_k$, then we have two cases. In the first, if $\sum_{k=i+1}^{i+2} bs(k) = \sum_{k=i+1}^{i+2} L_k$, this means that the size of the next two blocks is just enough to store the codewords of the next two symbols, so the next two blocks do not store the overflow of the i-th block. In the second, if $\sum_{k=i+1}^{i+2} bs(k) < \sum_{k=i+1}^{i+2} L_k$, this means that the next two blocks have an overflow, and its overflow is stored in the blocks following the $(i+2)$-th block. Hence, these two blocks do not store the overflow of the i-th block. In conclusion, if $Cumu(i,2) \leq 0$, we can conclude that the next two blocks do not store the overflow of the i-th block. Similarly, we continue to calculate $Cumu(i,3)$, $Cumu(i,4)$, $\cdots$, $Cumu(i,j)$ by enlarging j until we have $Cumu(i,j) > 0$. When $Cumu(i,j) > 0$ is encountered for the first time, according to the store rules, the last $Cumu(i,j)$ bits in the $(i+j)$-th block contains the overflow of the i-th block. Therefore, we concatenate the last $Cumu(i,j)$ bits in the $(i+j)$-th block to the end of B_i and update o_i. Precisely, let

$$\begin{aligned} &B_i \leftarrow B_i || \text{ the last } Cumu(i,j) \text{ bits in } B_{i+j}, \\ &o_i \leftarrow o_i + Cumu(i,j). \end{aligned} \tag{12}$$

where $||$ denotes the concatenation. Once the concatenation and update operations above are completed, o_i has the following two cases. In the first, if $o_i \geq 0$, this means that the overflow of the i-th block is completely found, thus we have $\overline{C}(B_i) \neq NULL$ and the i-th symbol is obtained. In the second, if $o_i < 0$, the remaining overflow of the i-th block is stored in the blocks following the $(i+j)$-th block. Therefore, we continue to calculate $Cumu(i+j,f)$ starting from $f = 1$ until we have $Cumu(i+j,f) > 0$. When $Cumu(i+j,f) > 0$, we perform concatenation and update operations. We repeat the above steps, and the repetition stops when $o_i \geq 0$. When $o_i \geq 0$, the decoding of the i-th symbol is achieved.

The above description shows that when we access the i-th symbol if the codeword length of the i-th symbol does not exceed the block size, i.e., $L_i \leq bs(i)$, the number of bits read to access the i-th symbol is $|p_i| + (L_i - |p_i|) = L_i$. Otherwise, we need to find the overflow of the i-th block from the blocks following the i-th block. In the latter, for any subsequent block k, we first read the prefix p_k to determine its codeword length L_k. By comparing L_k and $bs(k)$, we can determine whether the k-th block stores the overflow of the i-th block and if it does, we read the overflow; otherwise, we move to the next block. That is, we neither read the full codeword of the k-th symbol nor decode the k-th symbol such that the number of bits reads and the time required for access are reduced.

Algorithm 5 presents the details, and an example is given below to illustrate it. In Algorithm 5, Lines 2–4 handle the case where the codeword length does not exceed the block size, i.e., it can be directly decoded. Lines 5–16 handle the opposite. In particular, Lines 10–13 calculate the cumulative overflow variable in the following several blocks until $Cumu > 0$. We perform the concatenation and update operations in Lines 14–15. Finally, Line 16 completes the decoding of the i-th symbol.

Algorithm 5: Proposed access algorithm for canonical form codes

Input: An encoded sequence X, the average block size t, and an index i.
Output: The i-th symbol v.

```
1  Read the prefix p_i to determine L_i;
2  if L_i ≤ bs(i) then
3      B_i ← X[sp(i) : sp(i) + L_i − 1];
4      (v, r) ← C̄(B_i);
5  else
6      B_i ← X[sp(i) : sp(i + 1) − 1];
7      o_i ← bs(i) − L_i;
8      while o_i < 0 do
9          Cumu ← 0 ;
10         while Cumu ≤ 0 do
11             k ← the index of the next block;
12             read the prefix p_k to determine L_k;
13             Cumu ← Cumu + o_k;
14         B_i ← B_i|| the last Cumu bits in B_k;
15         o_i ← o_i + Cumu;
16     (v, r) ← C̄(B_i);
```

Example 4: Given a bit sequence $X = 0101110001101$, $N = 7$ and its codebook is shown in (9). The correspondence between prefix and codeword length is as follows: $\{0 \rightarrow 1, 10 \rightarrow 2, 11 \rightarrow 3\}$, where the left and right sides of the arrow denote the prefix that can determine the codeword length and the corresponding codeword length, respectively. And we obtain the start and end locations of each block by (2) and further get the sizes of these 7 blocks are 1, 2, 2, 2, 2, 2, 2, respectively. The steps to access the 3-th symbol are as follows. First, we read the prefix 11 in the 3-th block, which determines the codeword length is 3. However, the size of the 3-th block is 2, so we have $B_3 = 11$ and $o_3 = 2 - 3 = -1$. Second, we calculate the accumulation $Cumu(3,j)$ starting from $j = 1$. This accumulation stops when $Cumu(3,j) > 0$, thus we have $Cumu(3,2) = 1 > 0$. Then, we perform the concatenation and update operations and have $B_3 = 11||0 = 110$ and $o_3 = -1 + 1 = 0$. Finally, we decode sequence 110 to symbol c. The 3-th symbol in the source sequence is c.

5 Simulations and Analysis

In this section, we first give the simulations of the proposed method and three solutions with direct access capability. Second, we prove that when additional space is not allowed, the number of bits per access read in the proposed method is no more than that in the conventional method.

5.1 Simulations

We consider the direct access in a sequence of symbols, and the programs are written in C, compiled with gcc with optimization level -O3. We benchmarked the schemes on the platform equipped with Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz and 32 GB main memory on Ubuntu 18.04. To evaluate the proposed compression scheme, we choose seven different data benchmarks covering a variety of data

TABLE 3: Description of the datasets used

File Name	Alphabet Size	Input Size (Bytes)
book1	82	768,771
book2	96	610,856
alice19.txt	74	152,089
asyoulik.txt	68	125,179
dickens	100	10,192,446
nci	62	33,553,445
webster	98	41,458,703

types: Calgary and Canterbury Corpus[1], canterburycorpus[2], and silesia.[3] Table 3 presents some information about the data files involved, where the first two columns give the file name and the alphabet size. The last column shows the input file size in bytes.

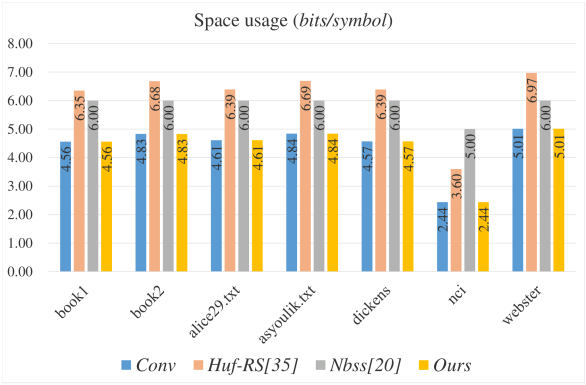


Fig. 5: Space usage evaluation ($bits/symbol$)

We implemented the algorithms as suggested by [20] (*Nbss*) and [35] (*Huf-RS*), as well as the conventional method (*Conv*), which encodes directly with Huffman coding. Since the codeword location of the i-th symbol is unknown on the Huffman-encoded stream, it needs to decode all preceding $i-1$ symbols to decode the i-th symbol. Among them, the underlying compression algorithm in our implementation of *Nbss* uses an optimal length-prefix compression by prepending a length field to a non-prefix code g, where g lists all symbols in decreasing probabilities. It maps them, starting with the most probable one, to binary sequences $\{0, 1, 00, 01, 10, 11, 000, 001, 010, 011 \cdots\}$ of increasing lengths. For instance, suppose $x_1, x_2, x_3, \cdots$ are all the symbols in decreasing probabilities, then x_1 is mapped to code 0, and x_5 is mapped to code 10, and so on. In contrast to prefix code, non-prefix code achieves a better compression ratio. Moreover, we prioritize storage efficiency by selecting the fixed block size in *Nbss*. Also, in our implementation of *Huf-RS*, the sampling frequency S is selected as 10, which is the best sampling ratio declared in [35].

Next, we give four simulations, and the details are as follows. First, we test the space usage for storing encoded streams in these four schemes: *Conv*, *Huf-RS*, *Nbss* and the proposed method (*Ours*). Among them, the prefix coding used in *Conv*, *Huf-RS*, and *Ours* are both Huffman coding. The space usage is measured in $bits/symbol$ and is computed by

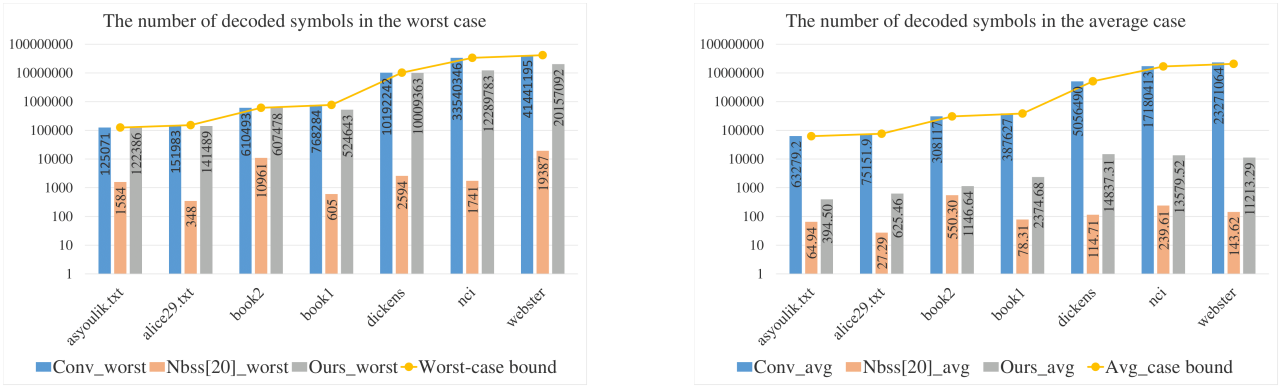
$$\text{Space usage} = \frac{|C(T)| + overhead(bits)}{N},$$

where the *overhead* denotes the size of additional space to store the encoded stream in bits. Figure 5 shows the results, and the values are labeled in the bars. It can be seen that the space cost of the proposed method is equal to that of the conventional method, and neither requires additional space to store the encoded stream. However, *Nbss* consumes, on average, at least one more bit per symbol than the proposed method, and its space usage mainly benefits from non-prefix codes. In other words, the space usage gap will be even larger if *Nbss* uses Huffman coding instead of non-prefix code as our proposed method. In addition, the space cost of *Huf-RS* scheme is about 40% more bits than the proposed method, and it has the worst space requirements in most cases.

We also test the number of decoded symbols in *Conv*, *Nbss* and *Ours* under the worst and average (abbreviated as avg) cases when we randomly access $10,000$ locations. The location is selected by $\text{mod}(rand(), N) + 1$, which returns a pseudo-random integral number in the range between 1 and N, where $\text{mod}(\cdot, \cdot)$ is the remainder operator. Figure 6 displays the simulation results. The values are labeled in the bars, and the bounds in the proposal are plotted with the results. The upper bounds of the proposed scheme in the worst and average cases are N and $\frac{N+1}{2}$ (see Section 5.2 for details), respectively. Figure 6 shows that when the encoded stream does not use additional space, the average number of decoded symbols in the proposed method is significantly less than that in the conventional method. In the worst case, sometimes, this number is close to that in the conventional method. This is because the blocks in the proposed method are cyclic, which may require decoding N symbols to access a symbol. Thus, the number of decoded symbols may be large in certain cases. This is also a price for the proposal to support direct access when additional space for storing encoded stream is not allowed. Correspondingly, with the help of additional data space, the number of decoded symbols in *Nbss* is less than that in the proposed method. However, *Nbss* has a hard requirement for additional space to store the encoded stream, this may not be available for some memory-constrained scenarios.

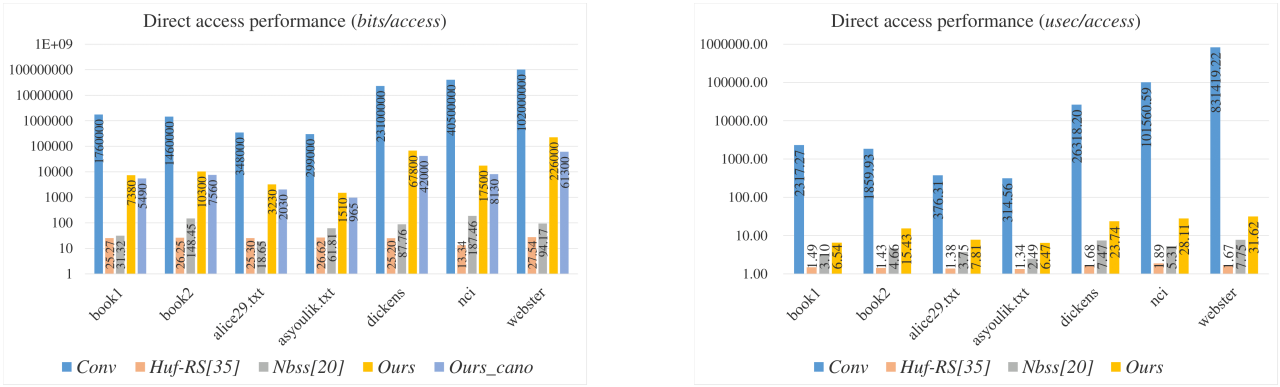
Next, we test the direct access performance of the above four solutions. All results are an average of $10,000$ runs on each encoded file. The direct access performance is measured in $bits/access$ and $usec/access$, which denote the number of bits per access read and the time per access consumed in a microsecond, respectively. Figure 7 gives the simulation results, and the values are labeled in the bars, where *Ours_cano* denotes the proposed method applied to canonical Huffman code (see Section 4 for details), and *Conv*, *Ours* and *Ours_cano* do not require additional space to store the encoded stream. In contrast, the solutions *Huf-RS* and *Nbss* require additional space (see Figure 5 for details). In terms of the number of bits read as shown in Figure 7(a), it can be seen that when the encoded stream

1. http://www.data-compression.info/Corpora/CalgaryCorpus/.
2. http://www.data-compression.info/Corpora/CanterburyCorpus/.
3. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia.



(a) (b)

Fig. 6: The number of decoded symbols when randomly accessing an item: (a) in the worst case and (b) in the average case



(a) (b)

Fig. 7: Direct access performance evaluation: (a) the number of bits read and (b) the time consumed

does not allow additional space, the bits read to access a symbol in the proposed method is above two orders of magnitude less than that in the conventional method. The compact property of canonical Huffman code can help reduce the number of bits read. When the encoded stream allows the use of additional space, *Huf-RS* reads fewer bits than *Nbss* in most cases, but as shown in Figure 5, *Huf-RS* has a higher space requirement. In terms of the time consumed as shown in Figure 7(b), if the additional space is not allowed, the proposed method takes significantly less time than the conventional method. If not, *Huf-RS* and *Nbss* perform better than the proposal, and *Huf-RS* benefits from the use of standard **rrr** structure to provide constant time queries. In conclusion, when the space overhead of storing the encoded stream is the key metric, the proposed method can achieve a good trade-off between space usage and access performance.

Finally, we suggest an improved version when a little auxiliary space is allowed. In this case, rather than applying the proposed rearranging method to the entire input data, we restrict it to operate on smaller chunks. Specifically, we first chunk the input data into equal-sized chunks, and the last chunk may be smaller. That is, if F is the number of symbols in the equal-sized chunks, then the last chunk has mod (N, F) symbols, where N is the total number of symbols in the input data and $\mathrm{mod}(\cdot,\cdot)$ is the remainder operator. Next, we apply the proposed rearranging method to each chunk individually. The encoded stream of each chunk is serialized into the final encoded stream, maintaining the original ordering. To support direct access, we require paying a small amount of additional space to record the indices of chunk boundaries in the final encoded stream. Consequently, to direct access the i-th symbol, we first retrieve the $\lceil i/F \rceil$-th chunk's boundary, then operate on the $\lceil i/F \rceil$-th chunk-encoded sequence to decode the i-th symbol. In this case, we decode at most F symbols to obtain the i-th symbol, reducing the number of bits read. Table 4 gives the direct access performance with $F = 10,000$, where the

additional space ratio for the input data T is defined as

$$\text{Additional space ratio} = \frac{\lfloor \frac{N}{F} \rfloor \times \lceil log_2 |C(T)| \rceil}{|C(T)|}, \quad (13)$$

where $|C(T)|$ is the final encoded stream length, and $\lfloor \frac{N}{F} \rfloor$ is the number of indexes recorded with the first index (referring to index 0) omitted. The larger ratio in the nic file is mainly because the nic file itself is a low entropy file, and its encoded length is quite large. From the comparison with Figure 7, it is observed that by paying a small price in storage space (approximately 0.057% additional storage space in the simulation), the number of bits per access read in the proposed method is reduced by about 90%. Indeed, the direct access performance in the improved version is tied to the chosen chunk size. A smaller chunk size results in a better access performance but at the cost of a higher additional space ratio. Table 5 tabulates the results when $F = 30$ is chosen. In Table 5, columns 2–4 list the additional space ratios for *Huf-RS*, *Nbss* and *Ours*. It can be seen that the proposed method has a lower additional space ratio than both *Huf-RS* and *Nbss*. Columns 5–8 give the direct access performance of *Huf-RS*, *Nbss*, *Ours* and *Ours_cano*. One can see that the proposed method can provide comparable performance to *Huf-RS*. Further, *Ours_cano* can sometimes even achieve better direct access performance than *Huf-RS*. In summary, when the improved version increases the proportion of additional space ratio (but still less than that in *Huf-RS* and *Nbss*), it can achieve direct access performance comparable to *Huf-RS* and *Nbss*.

5.2 Analysis

The cost of accessing a symbol is the total number of bits read. When the average number of bits in a block is denoted by t, decoding the i-th symbol by the conventional method requires reading i blocks, and each block has around t bits. Thus the cost is $i \times t$ bits. Similarly, in the proposed method, assuming that it reads η blocks to decode the i-th symbol, where $\eta \in \mathbb{N}^+$, thus the cost is $\eta \times t$ bits. Therefore, on average, comparing the number of bits read by the two methods can be approximated as the number of blocks they read.

Let X be a random variable, which denotes the number of blocks read to decode a symbol in the conventional method. Let $\mathbb{P}(X = i)$ denote the probability of decoding a symbol that requires reading i blocks, and the total number of symbols is N. Then, the expected value of the random variable X is expressed as

$$\mathbb{E}(X) = \sum_{i=1}^{N} \mathbb{P}(X = i) \times i. \quad (14)$$

Similar to the work in [30], we consider the case that each symbol has an equal probability of being decoded. Therefore,

$$\mathbb{E}(X) = \sum_{i=1}^{N} \frac{1}{N} \times i = \frac{N+1}{2}. \quad (15)$$

Let Y be a random variable, which denotes the number of blocks required to decode a symbol in the proposed method. To show that the proposed method reads fewer bits than the conventional method, it is necessary to prove that $\mathbb{E}(Y) \leq \mathbb{E}(X)$, i.e.,

$$\mathbb{E}(Y) \leq \frac{N+1}{2}. \quad (16)$$

Before formally proving (16), we first define a new structure called an envelope. Assuming that the proposed coding scheme requires reading the sequences in block i, block $i+1, \cdots$, block $i+\lambda$ to decode the i-th symbol, where $1 \leq i \leq N$, $\lambda \in \mathbb{N}$. Let $L(i) = i$, $H(i) = i + \lambda$ denote the start and end block indices when decoding the i-th symbol, respectively. Note that as blocks are cyclic, there may be $H(i) > N$, i.e., the $(N+i)$-th block and the i-th block refer to the same block. For any $i, j \in \mathbb{N}^+$, let $[i, j] := \{i, i+1, \cdots, j\}$, the definition of an envelope is given below.

Definition 1. *For any $j \in [i+1, N+i-1]$, if it does not exist*

$$L(i) < L(j) < H(i) < H(j), \quad (17)$$

then we define $\langle L(i), H(i) \rangle$ to be an envelope, and the set of block indices in the envelope can be expressed as $e_i = \{L(i), L(i)+1, \ldots, H(i)\}$, thus the length of $\langle L(i), H(i) \rangle$ is $H(i) - L(i) + 1$. It is worth noting that there may be an envelope consisting of a single block, i.e., $L(i) = H(i)$, which we call a closed envelope.

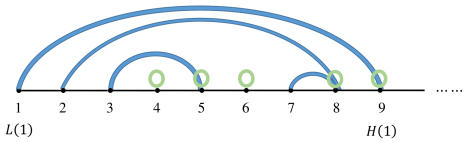


Fig. 8: An envelope example $\langle L(1), H(1) \rangle$

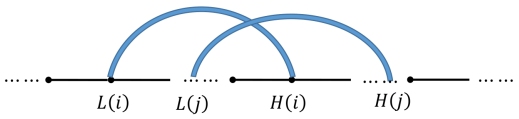


Fig. 9: $L(i) < L(j) < H(i) < H(j)$

In Figure 8, an envelope $\langle L(1), H(1) \rangle$ is shown, where $L(1) = 1$, $H(1) = 9$, the head and tail of the blue arc point to the block indices where decoding starts and ends, respectively, and the green circles represent the closed envelopes.

Based on the above definition, we prove below that in the proposed coding scheme, the $\langle L(i), H(i) \rangle$ obtained by decoding any symbol are all envelopes.

Lemma 1. *For a source sequence of N symbols, the $\langle L(i), H(i) \rangle$ obtained by the proposed coding scheme to decode the i-th symbol is an envelope, where $1 \leq i \leq N$, that is, for any $j \in [i+1, N+i-1]$, the inequality $L(j) < H(i) < H(j)$ does not hold.*

Proof. We prove this by contradiction. It is assuming $L(j) < H(i) < H(j)$ holds, such as in Figure 9. According to the neighbor-based storage scheme, the free space in block $H(i)$ should first store the overflow of the j-th codeword. If the overflow of the j-th codeword is all stored and there is still free space, the overflow of the i-th codeword can be stored there. Therefore, the end block index of decoding the j-th

TABLE 4: Direct access performance when $F = 10,000$ is chosen

File	Additional space ratio(%)	Direct access ($bits/access$)	
		Ours	*Ours_cano*
book1	0.0477	605.16	447.79
book2	0.0455	1015.12	744.35
alice29.txt	0.0428	328.44	108.94
asyoulik.txt	0.0396	125.90	37.79
dickens	0.0568	6073.30	4136.81
nci	0.1108	1645.43	907.06
webster	0.0560	17235.11	3720

TABLE 5: Direct access performance when $F = 30$ is chosen

File	Additional space ratio(%)			Direct access ($bits/access$)			
	Huf-RS [35]	*Nbss* [20]	*Ours*	*Huf-RS* [35]	*Nbss* [20]	*Ours*	*Ours_cano*
book1	39.25	31.58	16.08	25.27	31.32	24.13	19.72
book2	38.30	24.22	15.20	26.25	148.45	28.45	21.16
alice29.txt	38.61	30.15	14.45	25.30	18.65	16.10	12.52
asyoulik.txt	27.65	23.97	13.76	26.62	61.81	24.31	20.45
dickens	39.82	31.29	18.95	25.20	87.76	27.16	23.50
nci	47.54	104.91	36.92	13.34	187.46	12.23	10.16
webster	39.12	19.76	18.65	27.54	94.17	27.62	24.47

symbol cannot exceed $H(i)$; that is, $H(j) > H(i)$ does not hold. The proof is completed. □

Next, we define another structure called the maximum envelope.

Definition 2. *Given N envelopes $\{\langle L(j), H(j)\rangle\}_{j=1}^{N}$, if it does not exist $e_i \subsetneqq e_j$, where $\subsetneqq$ denotes proper subset. Then we call $\langle L(i), H(i)\rangle$ the maximum envelope.*

For the two envelopes $\langle L(i), H(i)\rangle$ and $\langle L(j), H(j)\rangle$, if $e_i \subsetneqq e_j$, we say that $\langle L(j), H(j)\rangle$ contains $\langle L(i), H(i)\rangle$. In the proposed scheme, by Definition 2, the envelope not contained in the other envelopes is maximum. Next, we define the corresponding maximum envelope $\langle L(m_i), H(m_i)\rangle$ of an envelope $\langle L(i), H(i)\rangle$. Specifically, if $\langle L(i), H(i)\rangle$ itself is a maximum envelope, let $\langle L(m_i), H(m_i)\rangle = \langle L(i), H(i)\rangle$; otherwise, according to the store rules and Lemma 1, there must be a larger envelope $\langle L(j), H(j)\rangle$ that contains $\langle L(i), H(i)\rangle$, where $\langle L(j), H(j)\rangle$ is larger than $\langle L(i), H(i)\rangle$ refers to $e_i \subsetneqq e_j$. For example, in Figure 8, we have $\langle L(1), H(1)\rangle$ larger than $\langle L(2), H(2)\rangle$. Furthermore, Definition 1 indicates that the total number of envelopes in the proposed scheme is N, thus among all envelopes larger than $\langle L(i), H(i)\rangle$, there must be the largest envelope. Let this largest envelope be the corresponding maximum envelope of $\langle L(i), H(i)\rangle$. Therefore, it can be seen that each envelope in the proposed scheme has a unique corresponding maximum envelope.

Below we show that in the proposed scheme, the N envelopes obtained by decoding can be seen as a union of γ disjoint maximum envelopes, where $\gamma \in [1, N]$.

Lemma 2. *For a source sequence of N symbols, all $\langle L(i), H(i)\rangle$ obtained in the proposed coding scheme, where $i \in [1, N]$, can be seen as a union of γ disjoint maximum envelopes, where $\gamma \in [1, N]$.*

Proof. As each envelope has a unique maximum envelope, we can first find the maximum envelope $\langle L(m_1), H(m_1)\rangle$ of $\langle L(1), H(1)\rangle$. Then, we try to find

$$\left\langle L(m_{H(m_1)+1}), H(m_{H(m_1)+1})\right\rangle$$

of $\langle L(H(m_1)+1), H(H(m_1)+1)\rangle$, and continue this step until all the maximum envelopes found contains the N envelopes. Therefore, these N envelopes can be seen as a union of γ disjoint maximum envelopes, where $\gamma \in [1, N]$. The proof is completed. □

The following theorem shows that the proposed method reads fewer bits than the conventional method, i.e., $\mathbb{E}(Y) \le \frac{N+1}{2}$.

Theorem 1. *In the proposed coding scheme, $\mathbb{E}(Y) \le \frac{N+1}{2}$ always holds.*

Proof. For a given source sequence of N symbols, suppose the lengths of the γ disjoint maximum envelopes in Lemma 2 are $\theta_1, \theta_2, \cdots, \theta_\gamma$, respectively, then we have $\sum_{i=1}^{\gamma} \theta_i = N$. Since in a maximum envelope of length θ_i, it requires reading at most $\theta_i - j + 1$ blocks to decode the j-th block within the maximum envelope, where $j \in [1, \theta_i]$. Therefore, decoding all blocks in a maximum envelope of length θ_i requires reading at most $\theta_i + (\theta_i - 1) + \cdots + 1 = \frac{(1+\theta_i)\times\theta_i}{2}$ blocks. Therefore, for the source sequence of N symbols, we have

$$\mathbb{E}(Y) \le \left(\frac{(1+\theta_1)\times\theta_1}{2} + \cdots + \frac{(1+\theta_\gamma)\times\theta_\gamma}{2}\right) \times \frac{1}{N} \tag{18}$$

$$= \frac{1}{2N} \times \left(N + \sum_{i=1}^{\gamma} \theta_i^2\right) \tag{19}$$

$$\le \frac{1}{2N} \times \left(N + \left(\sum_{i=1}^{\gamma} \theta_i\right)^2\right) \tag{20}$$

$$= \frac{N+1}{2}. \tag{21}$$

Note that the equality in (20) holds only when $\gamma = 1$. For $\gamma > 1, i \in [1, \gamma]$, as $\theta_i \ge 1$, (20) is always restrictedly less than (19). On average, the upper bound on the number of blocks that need to be read to access a symbol in the proposed scheme is $\frac{N+1}{2}$. The proof is completed. □

6 Conclusion

This paper proposes a rearranging method for prefix codes to support direct access to the encoded stream without requiring additional data space. Then, a novel lookup table construction method and fast decoding algorithm are presented. The simulation results show that when the encoded stream does not allow additional space, the number of bits per access read of the proposed method is above two orders of magnitude less than the conventional method. However, on average, the alternative solution consumes at least one more bit per symbol than the proposed method to support direct access. In other words, the proposed method can achieve a good trade-off between space usage and access performance. In addition, by paying a small amount of additional storage space (in the simulation, it is approximately 0.057%), the number of bits per access read in the proposed method can be significantly reduced by 90%. Furthermore, we show that the number of bits per access read of the proposed method can be further reduced for a canonical form code. Finally, we prove that the number of bits per access read in the proposed method is no more than that in the conventional method under the same amount of space usage.

References

[1] J. Abel and W. Teahan, "Universal text preprocessing for data compression," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 497–507, 2005.

[2] V. Anh and A. Moffat, "Improved word-aligned binary compression for text indexing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 6, pp. 857–861, 2006.

[3] N. Q. V. Hung, H. Jeung, and K. Aberer, "An evaluation of model-based approaches to sensor data compression," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 11, pp. 2434–2447, 2013.

[4] K. Zheng, Y. Zhao, D. Lian, B. Zheng, G. Liu, and X. Zhou, "Reference-based framework for spatio-temporal trajectory compression and query processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 11, pp. 2227–2240, 2020.

[5] W.-M. Lam and S. Kulkarni, "Extended synchronizing codewords for binary prefix codes," *IEEE Transactions on Information Theory*, vol. 42, no. 3, pp. 984–987, 1996.

[6] J. Cheng, "On the expected codeword length per symbol of optimal prefix codes for extended sources," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1692–1695, 2009.

[7] E. N. Gilbert and E. F. Moore, "Variable-length binary encodings," *Bell System Technical Journal*, vol. 38, no. 4, pp. 933–967, 1959.

[8] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[9] A. Fruchtman, Y. Gross, S. T. Klein, and D. Shapira, "Weighted adaptive huffman coding," in *2020 Data Compression Conference (DCC)*, 2020, pp. 368–368.

[10] Shang Xue and B. Oelmann, "Unary prefixed huffman coding for a group of quantized generalized gaussian sources," *IEEE Transactions on Communications*, vol. 54, no. 7, pp. 1164–1169, 2006.

[11] D. Baron and R. Baraniuk, "Faster sequential universal coding via block partitioning," *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1708–1710, 2006.

[12] N. Merhav, "Recent results in universal coding for probabilistic sources and individual sequences," in *Eighteenth Convention of Electrical and Electronics Engineers in Israel*, 1995, pp. 1.1.0/1–.

[13] S. T. Klein, T. C. Serebro, and D. Shapira, "Non-binary robust universal variable length codes," in *2020 Data Compression Conference (DCC)*, 2020, pp. 376–376.

[14] L. Farkas and T. Kói, "Universal random access error exponents for codebooks of different blocklengths," *IEEE Trans. Inf. Theory*, vol. 64, no. 4, pp. 2240–2252, April 2018.

[15] J. Choi, "Compressive random access with coded sparse identification vectors for MTC," *IEEE Trans. Commun*, vol. 66, no. 2, pp. 819–829, Feb 2018.

[16] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," *ACM Transactions on Information Systems (TOIS)*, vol. 18, no. 2, pp. 113–139, 2000.

[17] F. Transier and P. Sanders, "Engineering basic algorithms of an in-memory text search engine," *ACM Transactions on Information Systems (TOIS)*, vol. 29, no. 1, pp. 1–37, 2010.

[18] G. Jacobson, "Random access in Huffman-coded files," in *Data Compression Conference (DCC)*. IEEE, 1992, pp. 368–377.

[19] M. O. Külekci, "Enhanced variable-length codes: Improved compression with efficient random access," in *Data Compression Conference (DCC)*. IEEE, 2014, pp. 362–371.

[20] H. Zhou, D. Wang, and G. Wornell, "A simple class of efficient compression schemes supporting local access and editing," in *Information Theory (ISIT), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 2489–2493.

[21] N. R. Brisaboa, S. Ladra, and G. Navarro, "DACs: Bringing direct access to variable-length codes," *Information Processing & Management*, vol. 49, no. 1, pp. 392–404, 2013.

[22] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.

[23] M. O. Külekci, "Uniquely decodable and directly accessible non-prefix-free codes via wavelet trees," in *2013 IEEE International Symposium on Information Theory*. IEEE, 2013, pp. 1969–1973.

[24] A. Makhdoumi, S.-L. Huang, M. Médard, and Y. Polyanskiy, "On locally decodable source coding," in *2015 IEEE International Conference on Communications (ICC)*, 2015, pp. 4394–4399.

[25] V. Chandar, D. Shah, and G. W. Wornell, "A locally encodable and decodable compressed data structure," in *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2009, pp. 613–619.

[26] A. Mazumdar, V. Chandar, and G. W. Wornell, "Local recovery in data compression for general sources," in *2015 IEEE International Symposium on Information Theory (ISIT)*, 2015, pp. 2984–2988.

[27] K. Tatwawadi, S. S. Bidokhti, and T. Weissman, "On universal compression with constant random access," in *2018 IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 891–895.

[28] S. Vatedka, V. Chandar, and A. Tchamkerten, "O (log log n) worst-case local decoding and update efficiency for data compression," in *2020 IEEE International Symposium on Information Theory (ISIT)*, 2020, pp. 2371–2376.

[29] S. Vatedka and A. Tchamkerten, "Local decode and update for big data compression," *IEEE Transactions on Information Theory*, vol. 66, no. 9, pp. 5790–5805, 2020.

[30] A. Pananjady and T. A. Courtade, "The effect of local decodability constraints on variable-length compression," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2593–2608, 2018.

[31] F. Marcelloni and M. Vecchio, "A simple algorithm for data compression in wireless sensor networks," *IEEE Communications Letters*, vol. 12, no. 6, pp. 411–413, 2008.

[32] C. Tharini and P. V. Ranjan, "Design of modified adaptive huffman data compression algorithm for wireless sensor network," *Journal of Computer Science*, vol. 5, no. 6, p. 466, 2009.

[33] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.

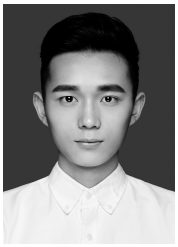
[34] S. T. Klein, "Space-and time-efficient decoding with canonical Huffman trees," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 1997, pp. 65–75.

[35] B. Adaş, E. Bayraktar, and M. O. Külekci, "Huffman codes versus augmented non-prefix-free codes," in *International Symposium on Experimental Algorithms*. Springer, 2015, pp. 315–326.

[36] J.-H. Jiang, C.-C. Chang, and T.-S. Chen, "An efficient huffman decoding method based on pattern partition and look-up table," in *1999 4th Optoelectronics and Communications Conference on Communications*, vol. 2, 1999, pp. 904–907.

Na Wang (Member, IEEE) received a Ph.D. degree in Information Science and Technology from the University of Science and Technology of China (USTC), Hefei, China. She is currently a researcher with the School of optoelectronic information and Computer Engineering at the University of Shanghai for Science and Technology (USST). Her research focuses on entropy coding algorithms and data compression.

Wei Yan (Member, IEEE) received the B.Sc. degree in mathematics and applied mathematics and the Ph.D. degree in Cyberspace Security from the University of Science and Technology of China (USTC), Hefei, China, in 2017 and 2022, respectively. He is currently a Researcher with the Theory Laboratory, 2012 Labs, Huawei Technologies Company Ltd. His research interest includes coding theory and data compression.



Hao Jiang received a B.Sc. degree in radio and television engineering from the Communication University of Zhejiang (CUZ), Hangzhou, China, in 2018. In 2021, he received M.Sc. degree in Information Science and Technology from the University of Science and Technology of China (USTC). He is currently a software developer with Alibaba Group, Hangzhou, China. His research focuses on data compression.

Sian-Jheng Lin (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 2004, 2006, and 2010, respectively. From 2010 to 2014, he was a Post-Doctoral Researcher with the Research Center for Information Technology Innovation, Academia Sinica. From 2014 to 2016, he was a Post-Doctoral Researcher with the Electrical Engineering Department at King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia. From 2016 to 2021, he was a Researcher with the School of Information Science and Technology at University of Science and Technology of China (USTC), Hefei, China. He is currently a Senior Scientist with the Theory Laboratory, 2012 Labs, Huawei Technologies Company Ltd. In recent years, his research focuses on the codes for storage systems and data compressions.

Yunghsiang S. Han (Fellow, IEEE) was born in Taipei, Taiwan, in 1962. He received B.Sc. and M.Sc. degrees in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1984 and 1986, respectively, and a Ph.D. from the School of Computer and Information Science, Syracuse University, Syracuse, NY, in 1993. From 1986 to 1988, he was a lecturer at Ming-Hsin Engineering College, Hsinchu, Taiwan. He was a teaching assistant from 1989 to 1992 and a research associate in the School of Computer and Information Science at Syracuse University from 1992 to 1993. From 1993 to 1997, he was an Associate Professor in the Department of Electronic Engineering at Hua Fan College of Humanities and Technology, Taipei Hsien, Taiwan. He was with the Department of Computer Science and Information Engineering at National Chi Nan University, Nantou, Taiwan from 1997 to 2004. He was promoted to Professor in 1998. He was a visiting scholar in the Department of Electrical Engineering at the University of Hawaii at Manoa, HI from June to October 2001, the SUPRIA visiting research scholar in the Department of Electrical Engineering and Computer Science and CASE center at Syracuse University, NY from September 2002 to January 2004 and July 2012 to June 2013, and the visiting scholar in the Department of Electrical and Computer Engineering at the University of Texas at Austin, TX from August 2008 to June 2009. He was with the Graduate Institute of Communication Engineering at National Taipei University, Taipei, Taiwan from August 2004 to July 2010. From August 2010 to January 2017, he was Chair Professor with the Department of Electrical Engineering at the National Taiwan University of Science and Technology. From February 2017 to February 2021, he was with the School of Electrical Engineering & Intelligentization at Dongguan University of Technology, China. Now he is with the Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China. He is also a Chair Professor at National Taipei University since February 2015. His research interests are in error-control coding, wireless networks, and security.

Dr. Han was a winner of the 1994 Syracuse University Doctoral Prize and a Fellow of IEEE. One of his papers won the prestigious 2013 ACM CCS Test-of-Time Award in cybersecurity.