SUMMARY In this paper, we propose two schemes, which enable any VF code to realize direct- or fast-access decoding for any source sequence. Direct-access decoding means that any source symbol of any position can be directly decoded within constant time, not depending on the length of source sequence \( N \), without decoding the whole codeword sequence. We also evaluate the memory size necessary to realize direct-access decoding or fast-access decoding with decoding delay \( O(\log \log N) \), \( O(\log N) \), and so on, in the proposed schemes.

**key words:** VF code, direct-access decoding, rank function, select function

### 1. Introduction

Fixed-to-variable length (FV) codes like Huffman codes and variable-to-fixed length (VF) codes like Tunstall codes are often used to store big data efficiently. But, since the FV codes have variable codeword length and the VF codes have variable parse length, we must decode the codeword sequence from the beginning even if we want to decode only one source symbol of a long source sequence \( x_1x_2\cdots x_N \). In the case of big data with very large \( N \), the decoding delay \( O(N) \) is not acceptable for the decoding of only one or a few source symbols.

In order to overcome this defect, direct-access decoding schemes have been studied such that any \( x_j \) can be decoded within constant time.

In the case of FV codes, several direct- or fast-access decoding schemes [1]–[7] have been proposed, which uses the so-called wavelet tree [1], [2], rank and/or select functions. Especially, in the case of Huffman codes, the direct-access decoding can be realized with the same coding rate asymptotically as the ordinary Huffman code if we use the same shape of wavelet tree as the Huffman code tree [2], [4].

For a binary sequence \( b = b_1b_2\cdots b_n \), rank function \( \text{rank}(b, l) \) and select function \( \text{select}(b, \ell) \) are defined as follows.

\[
\text{rank}(b, l) = \sum_{\ell=1}^{l} b_{\ell}, \quad (1)
\]

\[
\text{select}(b, \ell) = \min\{l : \text{rank}(b, l) = \ell\}. \quad (2)
\]

So, the rank function \( \text{rank}(b, l) \) gives the number of “1” included in the first \( l \) bits of \( b \), and the select function \( \text{select}(b, \ell) \) gives the position of the \( \ell \)-th “1” in \( b \). The rank and select functions can be calculated with constant time and \( n + o(n) \) memory space when the length of \( b \) is \( n \) [8]–[10].

On the other hand, few direct-access decoding schemes have been studied in the case of VF codes. Yoshida-Sasakawa-Sekine-Kida (YSSK) [11] proposed a direct-access decoding scheme such that bit \( b_j \) is assigned to each \( x_j \) of source sequence \( x = x_1x_2\cdots x_N \), and \( b_j \) is set as \( b_j = 1 \) if \( x_j \) is the last source symbol included in the same codeword, and \( b_j = 0 \) otherwise. YSSK scheme can be applied to any VF code. But, since YSSK scheme requires one bit \( b_j \) for each \( x_j, \) YSSK scheme is inefficient when the size of source alphabet is small. Especially, if the source alphabet is binary, YSSK scheme cannot attain any compression.

In this paper, in order to improve the above defect of YSSK scheme, we propose a modified YSSK scheme with a devised data structure such that after we divide both a source sequence and its codeword sequence into blocks, we apply YSSK scheme to the sequence of blocks. Although the modified YSSK scheme can attain good performance even for binary source sequences, the performance depends on how to implement the rank and select functions. So, by combining the idea of the modified YSSK scheme with the data structure proposed by Kimura-Suzuki-Sugano-Koike [7] to realize efficiently the rank function, we also proposed a self-contained scheme, which does not use rank and select functions, to realize the direct-access decoding or fast-access decoding with decoding delay \( O(\log \log N) \), \( O(\log N) \), and so on, for VF codes.

The modified YSSK scheme and the latter scheme are treated in Sects. 2 and 3, respectively.

The following notation is used in this paper.

**Notation**

- \( x \): a source sequence \( x = x_1x_2\cdots x_N \).
- \( x_j \): the \( j \)-th symbol of \( x \).
- \( y \): the sequence of codewords \( y = y_1y_2\cdots y_n \), which is encoded from \( x \) by a VF code.
- \( y_i \): the \( i \)-th codeword of \( y \). All \( y_i \) have the same length because of VF coding.
- \( N \): the length of \( x \).
- \( n \): the length of \( y \), i.e. the number of codewords included in \( y \).
- \( L_i \): the number of source symbols encoded into codeword \( y_i \).
In more detail, \( b_u \) and \( d_v \) can be obtained from
\( x = x_1 x_2 \cdots x_{\lfloor N/A \rfloor} \) and \( y = y_1 y_2 \cdots y_{n/B} \) by the following algorithm, where \( f_u(v) \) represents the index of the last source symbol included in \( y_v \). Note that \( f_u(v) \) and \( f_e(v) \) can easily be obtained when \( x \) is encoded into \( y \) sequentially.

**Algorithm 1 (Encoding):**

**Step1** Encode \( x \) into \( y = y_1 y_2 \cdots y_{n/B} \) by a VF code, and obtain \( f_u(v) \) and \( f_e(v) \).

**Step2** Set \( b_1 = 1, u = 2, v = 1 \).

**Step3** If \( f_u(v) \leq A(u-1) + 1 \leq f_e(v) \),
\[
b_u = 0,
\]
else (i.e., if \( f_e(v) < A(u-1) + 1 \)),
\[
b_u = 1, v = v + 1, d_v = A(u-1) + 1 - f_v(v).
\]

**Step4** If \( u = N/A \), exit, else, \( u = u + 1 \), go to Step3.

Note that the indexes \( j \) of the first and last source symbols included in \( y_v \) are given by \( X \times [\{ \lfloor b(v-1)/A \rfloor \} - d_{v-1} \} + 1 \) and \( A \times [\{ \lfloor b(v+1)/A \rfloor \} - d_{v-1} \} - d_{v+1} \), respectively. Hence, \( x_j \) can be decoded directly from \( y \), \( b = b_1 b_2 \cdots b_{n/B} \) and \( d = d_1 d_2 \cdots d_{n/B} \) as follows.

**Algorithm 2 (Direct-access decoding):**

**Step1** \( v = \text{rank}(b, \lfloor \frac{n}{A} \rfloor) \).

**Step2** If \( j > A \times [\{ b, v + 1 \rfloor - 1 \} - d_{v+1} \},
\[
\text{then } v = v + 1.
\]

**Step3** \( m_b = j - A \times [\{ b, v + 1 \rfloor - 1 \} + d_{v},
\]
\[
m_e = A \times [\{ b, v + 1 \rfloor - 1 \} - d_{v+1} \} - (j - 1).
\]

**Step4** \( x_j \) is the \( m_b \)-th source symbol obtained by decoding
\( y_v \) from the beginning, and \( x_j \) is also the \( m_e \)-th source symbol obtained by decoding \( y_v \) backward from the beginning.

As an example, assume that each \( y_j \) has \( L_j \) shown in Fig. 1. Then \( d_v \) and \( b_u \) are obtained by using Algorithm 1 as shown in Figs. 1 and 2, respectively. Furthermore, for instance, \( x_{76} \) can be directly decoded as follows.

**Example 1:** Assume that \( A = 5 \) and \( B = 4 \) are used, and \( L_i, d_v \) and \( b_u \) are given as shown in Figs. 1 and 2.

**Step1** For \( j = 76 \) and \( A = 5, v = \text{rank}(b, \lfloor \frac{n}{A} \rfloor) = \text{rank}(b, 16) = 7 \).

**Step2** \( x_{76} \) is included in \( y_7 \) since it holds that
\[
76 \leq 5 \times [\{ \lfloor b, 8 \rfloor - 1 \} - d_8 = 5[18 - 1] - 4 = 81.
\]

**Step3**
\[
m_b = j - A \times [\{ b, v - 1 \rfloor - 1 \} + d_v
\]
\[
= 76 - 5 \times [\{ b, 7 \rfloor - 1 \} + d_7
\]
\[
= 76 - 5 \times [15 - 1] + 1
\]
\[
= 7
\]
\[
m_e = A \times [\{ b, v + 1 \rfloor - 1 \} - d_{v+1} \} - (j - 1)
\]
\[
= 5 \times [\{ b, 8 \rfloor - 1 \} - d_8 - 75
\]
\[
= 5 \times [18 - 1] - 4 - 75
\]
\[
= 6
\]

**Step4** \( x_{76} \) is the 7-th source symbol obtained by decoding
\( y_v \) forward from the beginning, and \( x_{76} \) is also the 6-th
source symbol by decoding \(\hat{y}_v\) backward from the end.

In Step 4 of Algorithm 1, we need to decode at most \(B/2\) codewords \(y_i\) to obtain \(x_j\). But, decoding time does not depend on \(N\).

Next we consider the necessary memory size to store \(b\) and \(d\). The length of \(b\) is \(N/A\), and \(d_v\) satisfies \(0 \leq d_v \leq A-1\) and \(d_1 = 0\). Hence, the total memory size \(M_{\text{MYSSK}}\) is given by

\[
M_{\text{MYSSK}} = \frac{N}{A} + \frac{(n-B-1)\log A}{B}
\]

because \(N = nL\) and \(A < BL^{-1} < B\). Hence, by setting \(A\) a little large and setting \(B\) as \(A < BL^{-1}\), we can decrease the memory size considerably compared with the original YSSK scheme, which requires \(N\) bits.

It is worth noting that both YSSK scheme and the modified YSSK scheme require another memory space to store the data structure to calculate rank and select functions of \(b\) within constant time. Hence, the performance of these schemes depends on how to implement these functions.

### 3. Self-Contained Scheme

An efficient data structure to calculate rank function is proposed in [7]. So, by combining the data structures used in Sect. 2 and [7], we construct a self-contained scheme that requires neither the rank function nor the select function in this section.

In the modified YSSK scheme, \(y\) is divided into blocks \(\hat{y}_v\) with fixed length \(B\) independently from \(\hat{x}_v\). But, in this section, we divide \(y\) into blocks with variable length such that each block \(\hat{y}_v\) almost corresponds to each \(\hat{x}_v\), which has fixed length \(A\), where \(v = 1, 2, \ldots, N\). Then, we define \(\hat{y}_v\) such that the index \(i_v\) of the last codeword included in \(\hat{y}_v\) is given by

\[
i_v = \max\{i : \Gamma_i \leq A \cdot v\}.
\]

Now we define \(d_v^{(1)}\), which is the difference between \(A \cdot v\) and \(\Gamma_{i_v}\), as follows.

\[
d_v^{(1)} = A \cdot v - \Gamma_{i_v}.
\]

Then, for direct-access decoding we use \(\{i_v\}, \{d_v^{(1)}\}, v = 1, 2, \ldots, N/A\), which can easily be obtained in the encoding of \(x\). For simplicity, we set \(d_v^{(1)} = 0\).

For any \(j, x_j\) can be directly decoded from \(y, \{i_v\}\), and \(\{d_v^{(1)}\}\) by the following algorithm.

**Algorithm 3** (Direct access decoding):

**Step1** \(v = \lceil \frac{L}{A} \rceil\)

**Step2** If \(j > A \cdot v - d_v^{(1)}\), then \(v = v + 1\).

**Step3** \(m_v = j - [A \cdot (v - 1) - d_v^{(1)}]\), \(m_v = [A \cdot v - d_v^{(1)}] - j + 1\).

**Step4** \(x_j\) is the \(m_v\)-th source symbol obtained by decoding \(y\) from \(y_{i_v+1}, x_j\) is also the \(m_v\)-th source symbol obtained by decoding \(y\) backward from \(y_{i_v}\).

In order to obtain \(x_j\), we need to decode at most \(A/2\) codewords. On the other hand, the memory sizes to store \(\{i_v\}\) and \(\{d_v^{(1)}\}\) are shown in Table 1. Hence the total memory size is given by

\[
\frac{N}{A} \log \frac{N}{L} + \frac{N}{A} \log L^* < \frac{N}{A} \left( \log \frac{N}{L} + \log L^* + 2 \right)
\]

where \(N = nL\).

Although the above scheme is self-contained, it is not efficient because Eq. (7) is larger than Eq. (4). Therefore, we improve the performance by using a data structure similar to [7], i.e., we further divide each block \(\hat{y}_v\) into sub-blocks \(\hat{y}_{v,w}\) with variable length such that each sub-block \(\hat{y}_{v,w}\) almost corresponds of \(\hat{x}_{v,w}\), which is the sub-block of \(\hat{x}_v = \hat{x}_{v,1} \hat{x}_{v,2} \cdots \hat{x}_{v,A/C}\) and each \(\hat{x}_{v,w}\) has fixed length \(C\).

Then, if \(w = A/C\), the index of the last codeword included in \(\hat{y}_{v,w}\) is given by \(i_v\). For \(1 \leq w \leq A/C - 1\), we define \(\hat{y}_{v,w}\) such that the index of the last codeword included in \(\hat{y}_{v,w}\) is defined by \(i_{v-1} = i_{v,w}\) where \(\hat{y}_{v,w}\) is defined by

\[
i_{v,w} = \max\{l : \Gamma_{i_{v-1}+l} \leq A \cdot (v - 1) + C \cdot w, \quad 1 < i_{v-1} - i_{v-1}\}
\]

We also define \(d_v^{(2)}\) by

\[
d_v^{(2)} = A \cdot v - \Gamma_{i_{v,w}}
\]

\(\dagger\)For simplicity, we assume that \(A\) can be divided by \(C\).
Note that Eqs. (8) and (9) correspond to (5) and (6) in the one-stage division, respectively.

Then, for direct decoding, we use \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\), which can easily be obtained in the encoding of \(x\). For simplicity, we set \(d_{v}^{(1)} = 0\), and also note that \(d_{v}^{(1)} = d_{v}^{(2)}\).

Let \(i_{d}\) and \(i_{e}\) represent the indexes of the first and last codewords included in \(\hat{y}_{v,w}\), respectively, and let \(j_{d}\) and \(j_{e}\) represent the indexes of the first and last source symbol included in \(y_{v}\), and the last source symbol included in \(y_{w}\), respectively. Then, \(x_j\) can be directly decoded from \(y\), \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\) as follows.

Algorithm 4 (Direct access decoding):

**Step1** \(v = \lceil \frac{L}{A} \rceil\).

**Step2** If \(j > A \cdot v - d_v^{(1)}\), then \(v = v + 1, w = 1\), go to Step 5.

**Step3** \(w = \lceil \frac{j - A \cdot (v - 1)}{L} \rceil\).

**Step4** If \(j > A \cdot (v - 1) + C \cdot w - d_{v,w}^{(2)}\), then \(w = w + 1\).

**Step5**

\[i_{d} = i_{v-1} + i_{v,w-1} + 1,\]

\[j_{d} = A \cdot (v - 1) + C \cdot (w - 1) - d_{v,w-1}^{(2)},\]

\[m_{d} = j_{d} - 1,\]

\[i_{e} = i_{v} + i_{v,w},\]

\[j_{e} = A \cdot (v - 1) + C \cdot w - d_{v,w}^{(2)},\]

\[m_{e} = j_{e} - j + 1.\]

**Step6** \(x_j\) is the \(m_{d}\)-th source symbol obtained by decoding \(y\) from \(y_{i_{d}}\), and \(x_j\) is also the \(m_{e}\)-th source symbol obtained by decoding \(y\) backward from \(y_{i_{e}}\).

As an example, consider the same \(L_i\) as Fig. 1. Then, for \(A = 32, C = 8\), \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\) are obtained as shown in Fig. 3. Furthermore, for instance, \(x_{79}\) can be directly decoded as follows.

**Example 2:** Assume that \(A = 32, C = 8\) are used, and \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\) are given as shown in Fig. 3.

**Step1** \(v = \lceil \frac{79}{32} \rceil = 3\).

**Step2** \(x_{79}\) is included in \(\hat{y}_3\) since it holds that \(79 \leq 32 \times 3 - d_{3,1}^{(1)} = 96 - 4 = 92\).

**Step3** \(w = \lceil \frac{79 - 32 \times (3 - 1)}{8} \rceil = 2\).

**Step4** Since it holds that \(79 > 32 \times (3 - 1) + 8 \times 2 - d_{3,2}^{(2)} = 44\), \(x_{79}\) is included in \(\hat{y}_3\), and \(x_{79} = 79\).

**Example 3:** Assume that \(A = 32\) and \(C = 8\) are used, and \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\) are given as shown in Fig. 3.

**Step1** \(v = \lceil \frac{92}{32} \rceil = 3\).

**Step2** \(x_{92}\) is excluded from \(\hat{y}_3\), and \(x_{92} = \frac{92}{48} = 19\).

**Step3** \(w = \lceil \frac{92 - 32 \times (3 - 1)}{8} \rceil = 2\).

**Step4** Since it holds that \(92 > 32 \times (3 - 1) + 8 \times 2 - d_{3,2}^{(2)} = 44\), \(x_{92}\) is included in \(\hat{y}_3\), and \(x_{92} = 92\).

**Example 4:** Assume that \(A = 32\) and \(C = 8\) are used, and \(i_v\), \(i_{v,w}\), and \(d_{v,w}^{(2)}\) are given as shown in Fig. 3.

**Step1** \(v = \lceil \frac{79}{32} \rceil = 3\).

**Step2** \(x_{79}\) is included in \(\hat{y}_3\) since it holds that \(79 \leq 32 \times 3 - d_{3,1}^{(1)} = 96 - 4 = 92\).

**Step3** \(w = \lceil \frac{79 - 32 \times (3 - 1)}{8} \rceil = 2\).

**Step4** Since it holds that \(79 > 32 \times (3 - 1) + 8 \times 2 - d_{3,2}^{(2)} = 44\), \(x_{79}\) is included in \(\hat{y}_3\), and \(x_{79} = 79\).
Table 3

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log \frac{N}{\log N}$</td>
<td>constant</td>
<td>$O(N \log \log N)$</td>
</tr>
<tr>
<td>$\log \log N$</td>
<td>log log $N$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$(\log N)^{1+\alpha}$</td>
<td>$(\log N)^{\alpha}$</td>
<td>$O\left(\frac{N \log \log N}{\log N}\right)$</td>
</tr>
<tr>
<td>$N^b$</td>
<td>$bN^b$</td>
<td>$O(N^{1-b} \log N)$</td>
</tr>
</tbody>
</table>

Hence, in the case of the direct-access decoding with $C = \text{constant}$, the order of $M$ becomes

$$M = O(N \log \log N)$$  \hspace{1cm} (13)

if we use $A = O((\log N)/\log \log N)$. In the case of the fast-access decoding with $C = \log \log N$, $C = (\log N)\alpha$, or $C = bN^b$ for $a > 0$ and $1 \gg b > 0$, the order of $M$ can be further reduced to $O(N)$, $O\left(\frac{N \log \log N}{\log N}\right)$, $O(N^{1-b} \log N)$, respectively, as shown in Table 3.

4. Conclusion

In this paper, we proposed two schemes that enable the direct-access decoding for any VF codes. The first scheme is a modified scheme of [11], which uses rank and select functions. The second scheme is a self-contained scheme, which does not require rank and select functions.

References


