IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 3, 2020, accepted December 20, 2020, date of publication January 8, 2021, date of current version January 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3049868

Image Compression Using Chain Coding for Electronic Shelf Labels (ESL) Systems

YOUNGJUN KIM[1], KI-HYUNG KIM[2], (Senior Member, IEEE), AND WE-DUKE CHO[3]
[1]Department of Computer Engineering, Graduate School, Ajou University, Suwon 16499, South Korea
[2]Department of Cyber Security, Ajou University, Suwon 16499, South Korea
[3]Department of Electrical and Computer Engineering, Ajou University, Suwon 16499, South Korea

Corresponding author: Ki-Hyung Kim (kkim86@ajou.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology under Grant 2018R1D1A1B07048697, in part by the Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government Ministry of Trade, Industry and Energy (MOTIE) (The Competency Development Program for Industry Specialist) under Grant P0008703, and in part by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program, supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP), under Grant IITP-2020-2018-0-01396.

ABSTRACT One of the most important and recurring tasks in managing a store is to provide accurate, up-to-date price information to customers on the shelves. Manual updating of price tags has been a time-consuming, error-prone task with high labor costs. An Electronic shelf labels (ESL) system is becoming an attractive alternative for this task because of the dynamic-price-updating and customer's product-evaluation-display features. A common ESL system configuration in a retail store includes thousands of battery-powered ESL tags that are mostly connected wirelessly in a dense indoor environment. Raising the success ratio of wireless communication is essential for the system's viability due to its limited battery life. Most of the ESL traffic is the image data of goods that appear on the tags, and reducing the amount of the data is one of the most effective ways to enhance communication performance and reduce retransmission. This paper proposes an ESL image compression mechanism based on chain coding that utilizes ESL images' characteristics. The performance results show that the proposed mechanism could compress the ESL images smaller and decompress faster.

INDEX TERMS Electronic shelf label, chain code, image compression.

I. INTRODUCTION

The e-commerce market has grown rapidly due to improved Internet technology. Many online shopping sites have emerged, and product evaluation services such as reviews and star ratings are found on most of them. From the evaluation services, customers get much clearer information about the product they are considering buying, and it might lead to lower return rates. Also, the manufacturer could get feedback on the product and get an opportunity to improve it [2]. Providing evaluation data of products to customers at offline stores has been possible with Electronic Shelf Labels (ESL) systems.

ESL system could enhance the marketing competitiveness of the stores. It reduces price marking errors and the management-labor costs, especially for changing the price of goods on the way. The ability to change prices in real-time could allow retailers to adopt price strategies, such as changing prices based on the algorithms that consider competitor prices, supply and demand, and other external factors in the market. Because of these features, interests in ESL have been gradually increased around the world [3], [11].

A typical ESL system configuration in a retail store includes thousands of battery-powered ESL tags that are mostly connected wirelessly in a dense indoor environment. Raising the success ratio of wireless communication is essential for the system's viability due to its limited battery life [1]. Most of the ESL traffic is the image data of goods that appear on the tags, and reducing the amount of the data is one of the most effective ways to enhance communication performance and reduce retransmission. However, there are few studies on how to compress images used in ESL systems. The images are often compressed with typical compression algorithms such as Run-Length Encoding and deflate in the fields.

In this paper, we propose a new compression mechanism for compressing ESL images using chain coding. The

The associate editor coordinating the review of this manuscript and approving it for publication was Stefano Scanzio.

mechanism is called ECO; ESL Image Compression. It is specialized in compressing images consisting of simple figures and texts. The performance results show that our proposed compression algorithm could enhance the compression ratio and the decompression time than other algorithms. Increasing the compression ratio contributes to the reduction of traffic in wireless networks. A shorter decompression time means fewer computational loads on ESL devices, thereby prolonging ESL battery life.

The remainder of this paper consists as follows: We first describe the background knowledge required for the proposed mechanism in Section III. Next, describe the ECO image compression and decompression process in detail in Section IV. In Section V, we measure the performance of ECO and other compression algorithms for ESL images. Then, we present the evaluation results. Finally, we conclude the paper in Section VI.

II. RELATED WORKS

Lossless image compression algorithms can be classified as follows [5], [7]:

Run-Length Encoding (RLE) is an elementary form of lossless data compression that runs on sequences having the same value occurring many consecutive times, and it encodes the sequence to store only a single data value and its count. RLE can effectively compress data containing consecutive symbols, but results can be larger than the original data. An example of generalized RLE schemes is PackBits.

Arithmetic coding is a form of entropy encoding used in lossless data compression, in which the frequently seen symbols are encoded with fewer bits than lesser-seen symbols. JBIG2 is a monochrome image compression algorithm developed by JBIG(Joint Bi-level Image Experts Group) [24], and it is based on a form of arithmetic coding called the MQ coder, which is an adaptive binary arithmetic coder characterized by a multiplication-free approximation and a renormalization-driven update of the probability estimator [7]. JBIG2 supports both lossy and lossless compression modes, and the lossless mode can generally compress 3 – 5 times more than G4 [26].

Huffman coding [6] is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. The most frequent character matches the shortest code. The less frequent the character, the longer the code. Deflate is a lossless data compression algorithm that uses the LZSS [23] algorithm and Huffman coding. It is designed by Phil Katz and specified in RFC1951 [22]. The data format compressed by this algorithm consists of a series of blocks, which correspond to a series of blocks of input data. Each block is compressed using the LZSS algorithm and Huffman coding. CCITT Group 3 (G3) [25] 2-Dimensional (2D) and Group 4 (G4) [26] are also based on the Huffman coding. These are compression algorithms developed by CCITT (Consultative Committee on International Telegraphy and Telephony), a standard

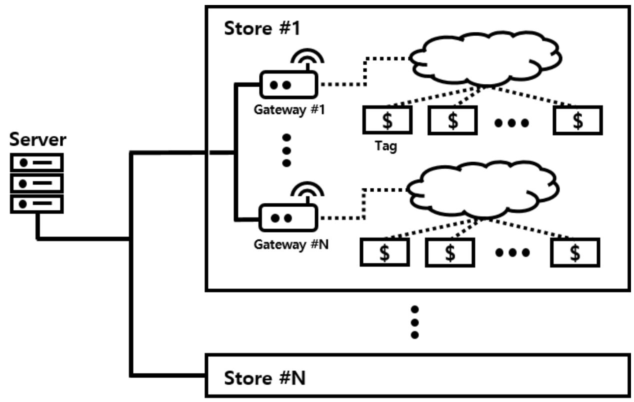


FIGURE 1. The structure of the ESL system network.

organization that has developed protocols for transmitting monochrome images to telephone lines and data networks. The encoding and decoding of G3 algorithms are fast and maintains a good compression ratio for various document data. Besides, encoded data includes data that G3 decoder can detect and correct errors. The G4 compresses monochrome images more efficiently. Data compressed into G4 is about half the size of data compressed into one-dimensional G3. The G4 is quite challenging to implement efficiently, but it encodes at least as fast as the G3, sometimes decoding faster than the G3. The G4 does not contain the synchronization code used to detect errors. These algorithms are non-adaptive and do not adjust the encoding algorithm to encode each bitmap with optimum efficiency.

Lempel-Ziv-Welch (LZW) is a widely known lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch [21]. The algorithm was published in 1984 as an improved version of the LZ78 algorithm published by Lempel and Ziv in 1978, and it is simple to implement and can achieve very high throughput when implemented with hardware. The main operating principle is to register a recurring bit sequence in the dictionary and replace the repeated pattern with the dictionary's index code. The generated dictionary does not need to be sent to the decoder, and it is reconstructed during the decoding process.

III. BACKGROUND

In the background section, we firstly describe the overall ESL system architecture on which the proposed algorithm targets. Then, we describe the three image processing schemes employed in the proposed algorithm in the next section.

A. ESL SYSTEM

The ESL system consists of servers, gateways and tags as shown in Figure 1 [10]– [12]. The server manages all ESL system information, such as the product information displayed on each tag and a list of tags that communicate with each gateway. The gateway forms the wireless network and manages the tags that have joined the network. It also acts as an intermediary between the server and tags, supporting

FIGURE 2. Example of ESL tags.

FIGURE 3. A example of MTFT.

communication between them. The tag receives the image from the gateway and displays it on the screen. It works in synchronization with the gateway. When the tag has no more work to do, it goes into sleep mode to minimize power consumption.

The image displayed on the tag is designed to communicate the product information to customers effectively. In general, the image consists of a small number of colors. Usually, two colors (black, white) or three colors (black, white, red, or yellow) are used as shown in Figure 2, and most areas of the image are text. Product information such as product name, price, and reviews of customers is displayed clearly. Different text colors or background colors might emphasize the information. The greater the number of colors that an e-paper module can express, the higher its price and power consumption.

B. MOVE-TO-FRONT TRANSFORM

Move-to-front Transform (MTFT) is a method to reduce the entropy of data and compress it efficiently [14], [15]. This algorithm replaces the input data with the index, and the recently used symbol is moved to the beginning of the stack. For example, if "ABBBCBBBBB" is entered in the MTFT, it is processed as shown in Figure 3, resulting in conversion to "01011021000". Meanwhile, MTFT is used in conjunction with Burrows-Wheller Transform (BWT) to reduce the entropy more efficiently [17], [18].

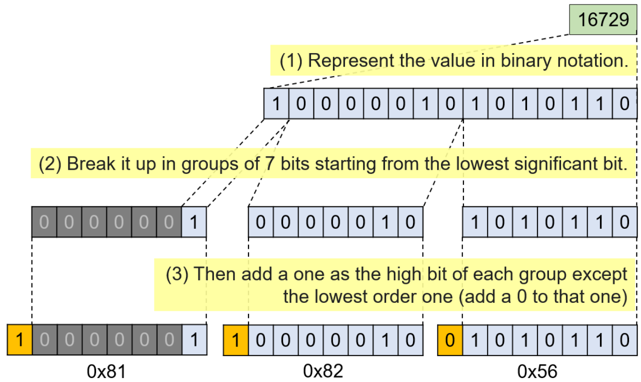


FIGURE 4. Diagram showing how to convert 16,729 from decimal to base-128 representation.

C. CHAIN CODING

Chain coding encodes the direction of connection between pixels that form the boundary of an image object. Encoded chain code is typically compressed using algorithms such as RLE or Huffman Coding. Freeman's 8-directive chain code (FCCE), which first appeared, represents the eight adjacent directions of the pixels in each symbol [16]. Since the higher the number of symbols, the greater the number of bits required to encode each symbol, then the chain code algorithms with fewer symbols appear. As one of them, Vertex Chain Code (VCC) [19], which uses three symbols, encodes the number of adjacent boundary pixels at the vertex. In another algorithm, Three OrThogonal symbol chain code (3OT) [20] determines the symbol according to the direction of the previous encoding direction. The ECO uses three symbols, and details are covered in Section IV-B2.

D. VARIABLE-LENGTH INTEGER ENCODING

Variable-length integers encoding (VarInt) is an algorithm to compress fixed-length integers into variable-length integers to save space. Base-128 is a kind of these algorithms. Figure 4 shows how to convert 16,729 from decimal to base-128 representation.

IV. PROPOSED MECHANISM

The main target of the mechanism is the images used in ESL system. The images consist of text, barcodes, and simple figures. Text usually accounts for a large portion of the images. Also, there are two or three colors used in the images.

In this section, the data structure, the compression, and decompression of ECO are described in detail, and Figure 5 is used to help explain these processes.

A. DATA STRUCTURE

The compressed file format of ECO is shown in Figure 6. α is the length of data bits in the variable-length integer encoding described in Section III-D. It consists of 2 bits and represents a value in the range of 1 to 4, as shown in Table 1. The width and height of the image are expressed in VarInt format. There are three types of chain code table items, and depending on

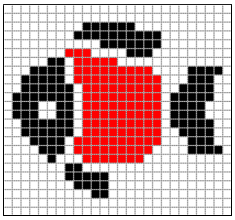


FIGURE 5. A image for testing the proposed mechanism (26 × 24px).

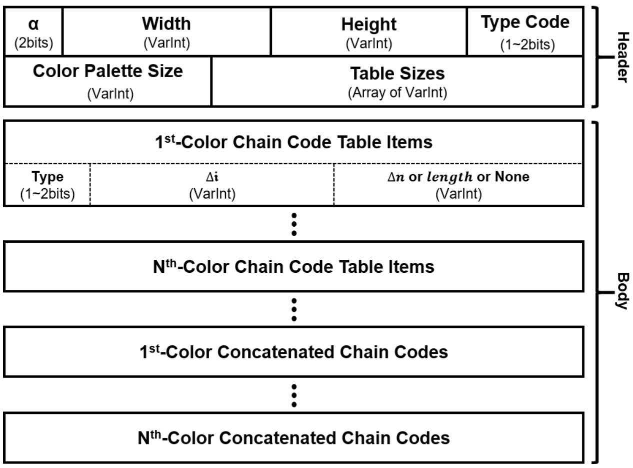


FIGURE 6. Structure of image file compressed with ECO.

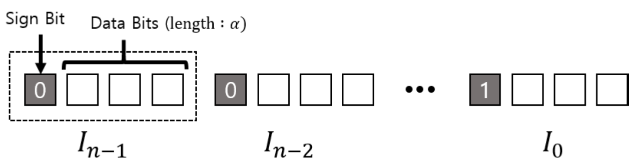


FIGURE 7. Variable-length Integer structure.

TABLE 1. Bit representation of α.

α	Bit Representation
1	00
2	01
3	10
4	11

TABLE 2. Bit representation of type code.

The most frequent type	Bit Representation
A	0
B	10
C	11

which type is the most, *Type Code* is represented in 1 or 2 bits, as shown in Table 2. The description of the types is covered in Section IV-B3.

Color palette is the color range of an image. White is the default color, and it is not counted in the size of the color palette. For example, the color palette size in Figure 5 is 2. The color palettes of the images used in this paper are shown in Table 3. The color palette size of an image is expressed in VarInt format.

TABLE 3. Color palettes for image type.

Type	Size	Colors
Monochrome	1	Black
3-color	2	Black, Red
8-color	7	Black, Red, Blue, Orange, Magenta, Green, Yellow

TABLE 4. Examples of VarInt encoding.

Decimal	Encoded as	Decimal	Encoded as
1	10	63	000000000010
2	11	126	010101010111
3	0010	127	00000000000010
6	0111	254	01010101010111
7	000010	255	0000000000000010
14	010111	510	0101010101010111
15	00000010	511	000000000000000010
30	01010111	1022	010101010101010111
31	0000000010	1023	00000000000000000010
62	0101010111	2046	01010101010101010111

Table Sizes is a VarInt array that stores the number of chain code table items by color. Because the number of table items can be 0, the value plus one is stored.

Many fields in Figure 6 are encoded in the VarInt format. The form of VarInt used in this paper consists of a series of chunks comprising a sign bit and data bits, as shown in Figure 7. If not the last chunk, its sign bit is 0; otherwise, it is 1. The α-length data bit can represent an integer of 2^α, and the range of integers a single chunk can express is Equation 1. The integer used in ECO is always greater than 0, the minimum value of I_i is 1. An integer encoded in VarInt can be decoded using Equation 2. Table 4 is an example of a VarInt representation of decimal values.

$$1 \le I_i \le 2^\alpha \quad (0 \le i < n) \tag{1}$$

$$\sum_{i=0}^{n-1} I_i \cdot 2^{\alpha i} \tag{2}$$

B. COMPRESSION

The compression process is carried out in four stages, as shown in Figure 8. The number of outputs for each stage is N, the size of an image's color palette.

1) COLOR SEPARATION AND BINARY MOVE-TO-FRONT TRANSFORM

In this stage, an image is separated into several mono-color images - one image for each color portion of the image. Subsequently, these mono-color images are converted into binary sequences: each pixel in the image changes to 0 if

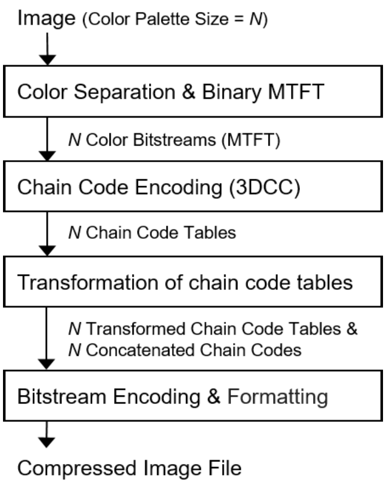


FIGURE 8. 4 stages of compression process.

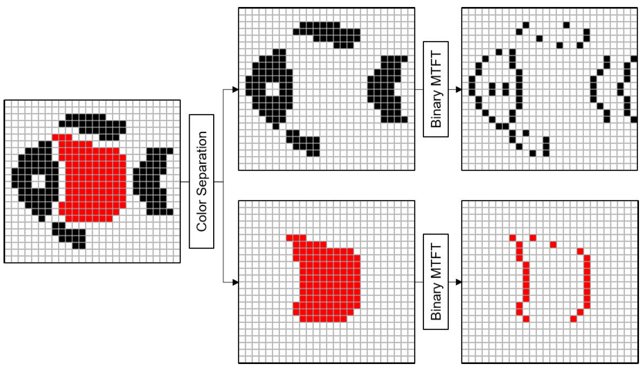


FIGURE 9. Color separation and binary MTFT.

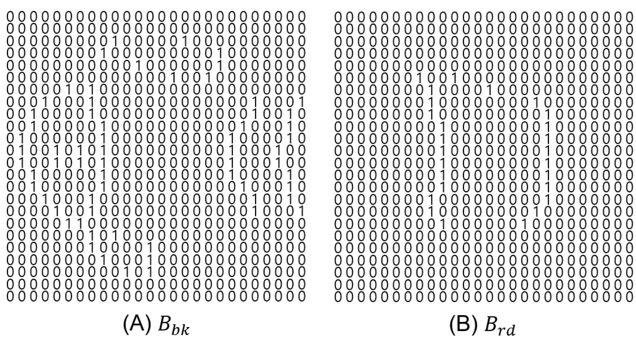


FIGURE 10. Two MTF-transformed bitstreams (B_{bk}, B_{rd}).

it is white or 1 if it is not. Then, Binary MTFT is applied to the sequences: the MTFT used in this paper is named binary MTFT because it deals only with the sequences consisting of two symbols. Figure 9 illustrates the process of change in Figure 5 at this stage. A pseudo algorithm of the process is shown in algorithm 1. The algorithm performs color separation and binary MTFT in one cycle. The two bitstreams in Figure 10 result from performing the algorithm in Figure 5.

Algorithm 1 Color Separation and Binary MTFT

```
Input  : A pixel array P of an image
Output: A array C[0..N] of bitstreams
/* N is color palette size                */
prev ← −1
C[0..N] ← 0
for i ← 0 to P.length do
    idx ← Index of P[i] in color palette
    for j ← 0 to N do
        if j is not idx then
            if prev = j then
                C[j][i] ← 1
        else if prev is not j then
            C[j][i] ← 1
    end
    prev ← idx
end
```

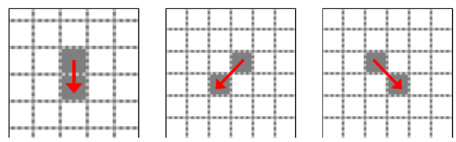


FIGURE 11. Downward directions (Symbol: 0, 1, 2).

2) DOWNWARD DIRECTION CHAIN CODE (DCC)

The bitstreams of the previous stage are encoded using a chain code in this stage. The chain code encodes only three directions, as shown in Figure 11. Because the direction of progress of the encoding is from top-left to bottom-right of an image, only the downward directions are needed for the chain code. After running binary MTFT, the probability of encoding to the pixel's left and right directions is very low. A pseudo algorithm of the encoding process is shown in algorithm 2. Table 5 is the chain code table that results from applying the algorithm to the previous stage results.

3) TRANSFORMATION OF CHAIN CODE TABLES

This section covers the preparatory stage for converting the chain code tables of the previous stage into bitstreams.

Initially, each item on the tables is classified according to Table 6. For example, The codes of the 2nd, 4th, 5th, 6th, 13th and 15th items in CT_{bk} are empty. Thus, the type of these items is A. Similarly, the type of the 2nd and 3rd in CT_{rd} is also A. The code of the 12th item of CT_{bk} is the same as that of the 11th. So, the type of it is B. The type of the others is C.

Secondly, the Δi of each item in the chain code tables is calculated using Equation 3. This equation ensures that Δi is always an integer greater than zero.

$$\Delta i_n = \begin{cases} i_n + 1, & \text{if } n = 1 \\ i_n - i_{n-1}, & \text{otherwise} \end{cases} \tag{3}$$

Algorithm 2 Downward Direction Chain Code

```
Input  : A bitstream B of a monocolor image
Output: Chain code table T
/* W is the width of the image */
c ← 0
L ← B.length
for i ← 0 to L do
    if B[i] is 1 then
        B[i] ← 0
        T[c].i ← i
        T[c].code ← empty
        j ← i
        while j < L do
            K ← j + W
            if K < L and B[K] = 1 then
                B[K] ← 0
                T[c].code = T[c].code + 0
                j ← K
            else if K − 1 < L and B[K − 1] = 1 then
                B[K − 1] ← 0
                T[c].code = T[c].code + 1
                j ← K − 1
            else if K + 1 < L and B[K + 1] = 1 then
                B[K + 1] ← 0
                T[c].code = T[c].code + 2
                j ← K + 1
            else
                break
        end
        c ← c + 1
end
```

Thirdly, the Δn of the B type items are calculated, and the *Length* of the code of C type items are measured. The Δn is the number difference between an item and the previous item with the same code. Through the steps so far, TT_{bk} and TT_{rd} are obtained from CT_{bk} and CT_{rd}, as shown in Table 7.

Lastly, The codes of the C type items are combined into one. For example, the concatenated chain codes (CC_{bk}, CC_{rd}) obtained from CT_{bk} and CT_{rd} are as follows:

$$CC_{bk} = 10100200000101202101002022220110002201\ 00102002000$$

$$CC_{rd} = 2002000001022000000011$$

4) BITSTREAM ENCODING AND FILE FORMATTING

This section covers the final stage of the compression process. The transformed chain code tables and the concatenated chain codes are encoded as bitstreams, and the results are formatted as shown as Figure 6. In the example of this stage, the α is 1.

TABLE 5. Chain code tables of two bitstreams (CT_{bk}, CT_{rd}).

No.	i	Code
1	61	1
2	67	-
3	96	01
4	115	-
5	144	-
6	161	-
7	163	00200000101202
8	185	1010020222
9	203	011000220
10	207	100102002
11	290	0
12	292	0
13	477	-
14	506	00
15	556	-

No.	i	Code
1	137	200200000102
2	140	-
3	169	-
4	199	2000000011

TABLE 6. Item types.

Type	Condition
A	The length of the code is 0
B	Some of the previous items have the same code
C	Otherwise

TABLE 7. Transformed chain code tables (TT_{bk}, TT_{rd}).

No.	Type	Δi	Length	Δn	Encoded
1	C	62	1	-	0 0101010111 10
2	A	6	-	-	10 0111
3	C	29	2	-	0 01010110 11
4	A	19	-	-	10 00010010
5	A	29	-	-	10 01010110
6	A	17	-	-	10 00000110
7	C	2	14	-	0 11 010111
8	C	22	10	-	0 00010111 000111
9	C	18	9	-	0 00000111 000110
10	C	4	9	-	0 0011 000110
11	C	83	1	-	0 000100010010 10
12	B	2	-	1	11 11 10
13	A	185	-	-	10 00010101000110
14	C	29	2	-	0 01010110 11
15	A	50	-	-	10 0100000111

No.	Type	Δi	Length	Encoded
1	C	138	12	0 00000001000111 010011
2	A	3	-	10 0010
3	A	29	-	10 01010110
4	C	30	10	0 01010111 000111

a: ENCODING TRANSFORMED CHAIN CODE TABLES AS BITSTREAMS

Each item in the transformed chain code tables is encoded as bitstreams, in order of *Type*, Δi, Δn, or *Length*. Before

TABLE 8. Chain code bit transformation table.

Prev \ Current	0	1	2	R
0	0	10	11	-
R	-	0	1	-
Other	0	110	111	10

encoding *Type*, it is necessary to identify which type of item has the most frequent. The most frequent type is encoded as 0. The following type in alphabetical order is encoded as 10, and the other is encoded as 11. Subsequently, Δi, Δn and *Length* are encoded as VarInt. For example, The most frequent type of item on Table 7 is C, so this type is encoded as 0, A type is encoded as 10, and B type is encoded as 11. As a consequence, the table is encoded as follows:

$$\begin{aligned} E(TT_{bk}) = {} & 001010101111010011100101011011100001 \\ & 001010010101101000000110011010111100001 \\ & 0111000111000000011100011000011000110 \\ & 00001000100101011111010000101010001 \\ & 1000101011011100100000111 \\ E(TT_{rd}) = {} & 00000000100011101001110001010 \\ & 01010110001010111000111 \end{aligned}$$

b: ENCODING CONCATENATED CHAIN CODES TO BITSTREAMS

The concatenated chain codes are encoded as bitstreams using Table 8. The encoding result of the current symbol depends on the previous symbol. The series of symbol 0 is abbreviated to symbol R if its length is longer than R_{min}. R_{min} can be obtained as Equation 4.

$$R_{min} = \alpha + 3 \tag{4}$$

The α used in the example is 1, so R_{min} is 4. Therefore, the series of symbol 0 in the length of more than four is abbreviated. Meanwhile, behind the symbol R comes the VarInt encoded difference between the length of the symbol 0 series and the R_{min}. There is only one series in CC_{bk} that is over four in length, and its length is five. That is consequently encoded in 1010. CC_{bk} and CC_{rd} are finally encoded as bitstreams as follows:

$$\begin{aligned} E(CC_{bk}) = {} & 1100100011101000101110111100100011011 \\ & 1111110101100001111101000100110011000 \\ E(CC_{rd}) = {} & 1110011101000111111000100110 \end{aligned}$$

c: FILE FORMATTING

This step creates a header and associates it with the bitstreams in the previous stage to make a compressed file as Figure 6. For example, the header of Figure 5 and the compressed result is as follows. It is 358 bits in size, about 45 bytes.

$$\begin{aligned} & Header \\ & = \alpha + VarInt(Width) + VarInt(Height) \\ & \quad + TypeCode + VarInt(ColorPaletteSize) \\ & \quad + VarInt(ThesizeofTT_{bk} + 1) \\ & \quad + VarInt(ThesizeofTT_{rd} + 1) \\ & = 00 + 01000111 + 01000011 \\ & \quad + 11 + 11 \\ & \quad + 00000011 \\ & \quad + 0110 \\ & = 00010001110100001111110000000110110 \end{aligned}$$

$$\begin{aligned} & Comp(Figure\ 5) \\ & = Header + E(TT_{bk}) + E(TT_{rd}) + E(CC_{bk}) + E(CC_{rd}) \\ & = 000100011101000011111100000011011000101010111 \\ & \quad 10100111001010110111000010010100101011010000 \\ & \quad 00110011010111000010111000111000000111000110 \\ & \quad 0001100011000001000100101011111101000010101000 \\ & \quad 1100010101101110010000011100000000010001110100 \\ & \quad 111000101001010110001010111000111110010001110 \\ & \quad 100010111011110010001101111111101011000011111 \\ & \quad 010001001100110001110011101000111111000100110 \end{aligned}$$

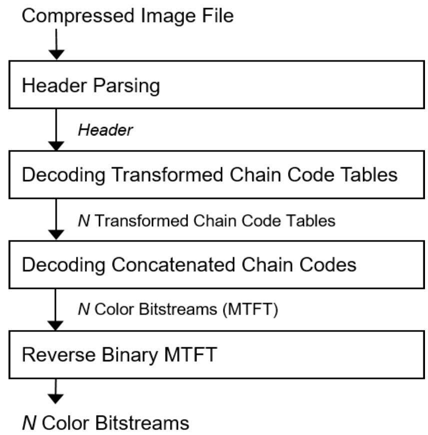


FIGURE 12. 4 stages of decompression process.

C. DECOMPRESSION

The decompression process is carried out in four stages, as shown as Figure 12.

1) HEADER PARSING

At this stage, the header part of the compressed image file is parsed. The headers of the $Comp(Figure\ 5)$ are parsed in the following order: (1) Obtain the value of α from the first two bits. These two bits are "00", so α is 1. (2) Parse *Width* and *Height*. These are encoded as VarInt. So, split the data into a chunk unit and read them until the first bit of a chunk is 1. As a result, *Width* is "01000111", and *Height* is "01000011". Decoding these two values using Equation 2 results in 26 and 24. (3) Read two bits to find out the most frequent type of chain code table items. These two bits are

Algorithm 3 Decoding Concatenated Chain Codes With a Transformed Chain Code Table

```
Input  : A transformed chain code table TT
Output: A bitstream B
/* W is the width of the header      */
/* H is the height of the header     */
B[0..W ∗ H] ← 0
idx ← −1
for i ← 0 to TT.length do
    item ← TT[i]
    idx ← idx + item.Δi
    B[idx] ← 1
    if item.type is A then
        continue
    else if item.type is B then
        code ← TT[i + item.Δn].code
        length ← TT[i + item.Δn].length
    else if item.type is C then
        item.code ← getChainCodes(item.length)
        code ← item.code
        length ← item.length
    c ← idx
    for j ← 0 to length do
        if code[j] is 0 then
            c ← c + W
            B[c] ← 1
        else if code[j] is 1 then
            c ← c + W − 1
            B[c] ← 1
        else if code[j] is 2 then
            c ← c + W + 1
            B[c] ← 1
    end
end
```

```
00000000000000000000000000    00000000000000000000000000
00000000000000000000000000    00000000000000000000000000
00000000011111100000000000    00000000000000000000000000
00000000111111111100000000    00000000000000000000000000
00000000000111111100000000    00000000000000000000000000
00000000000000111000000000    00000001110000000000000000
00000110000000000000000000    00000000111110000000000000
00011110000000000000011110    00000000111111111000000000
00111110000000000000011100    00000000111111111100000000
00111111100000000000111100    00000000011111111100000000
01111111100000000001111100    00000000011111111100000000
01110011000000000001111000    00000000011111111100000000
01110011000000000001111000    00000000011111111100000000
00111111100000000001111100    00000000011111111100000000
00111111100000000000111100    00000000011111111100000000
00011110000000000000011100    00000000111111111100000000
00001110000000000000011110    00000000111111111000000000
00000100000000000000000000    00000000011111110000000000
00000001100000000000000000    00000000000000000000000000
00000001111100000000000000    00000000000000000000000000
00000000111100000000000000    00000000000000000000000000
00000000001100000000000000    00000000000000000000000000
00000000000000000000000000    00000000000000000000000000
00000000000000000000000000    00000000000000000000000000
```

(A) O_{bk} (B) O_{rd}

FIGURE 13. Decompression results (O_{bk}, O_{rd}).

"11", so the most frequent type is C. This value is used when decoding chain code tables. (4) Parse the color palette size encoded as VarInt. The encoded value is "11", so the color palette size is 2. (5) Read the table-size as many times as the color palette size. In this case, the color palette size is 2, so two table-sizes are read. The two read values are "00000011" and "0110", and these are decoded into 16 and 5. The values reduced by 1 to these are the size of the chain code tables. As a result, the size of the first chain code table is 15, and the size of the second chain code table is 4.

Algorithm 4 Reverse Binary Move-to-Front Transform

```
Input  : A bit array B
Output: A bit array B
p ← 0
for i ← 0 to B.length do
    b ← B[i]
    if b is 1 then
        if p is 1 then
            p ← 0
        else
            p ← 1
    B[i] ← p
end
```

Algorithm 5 Reverse Binary Move-to-Front Transform Using Transition Table

```
Input  : A byte array B
Output: A byte array B
T = Transition table of two dimensional array format (2
× 256)
p ← 0
for i ← 0 to B.length do
    B[i], p ← T(B[i], p)
end
```

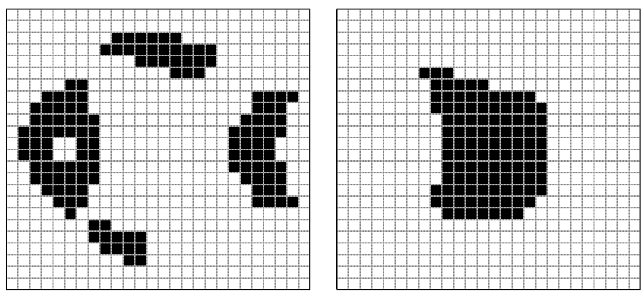


FIGURE 14. Images of decompression results (O_{bk}, O_{rd}).

2) DECODING TRANSFORMED CHAIN CODE TABLES

At this stage, the transformed chain code tables are decoded. To get a transformed chain code table, read the transformed chain code table item repeatedly as its table size in the header. For example, since we knew the size of the first and second tables in the previous stage, parse items as much as the table size from the remaining data R_{stage_1} as shown below.

$$R_{stage_1} = 0010101011110100111001010110111000010010$$

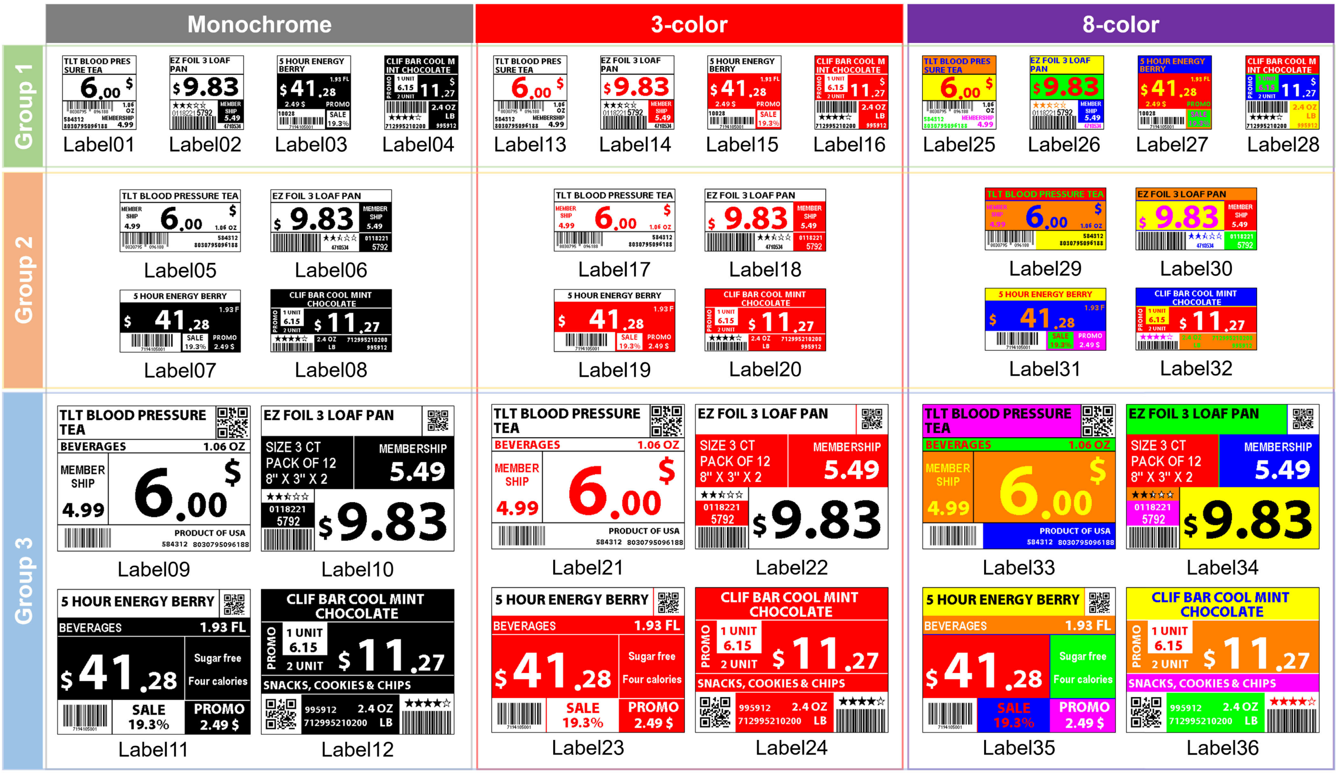


FIGURE 15. The ESL images for the performance evaluation.

$$
\begin{aligned}
&1001010110100000011001101011100001011100\\
&0011100000011100011000011000110000001000\\
&100101011111010000101010001100010101110\\
&11100100000011100000000010001110100111000 1\\
&010010101100010101110001111100100011110\\
&1000101110111110010001101111111101011000\\
&011111010001001100110001110011101000111\\
&1111000100110
\end{aligned}
$$

The first item of the first transformed chain code table is decoded as follows. (1) The bit part of *Type* is "0". So the most frequent type is C. (2) Parse the Δi encoded as VarInt. It is "0101010111". Therefore, the Δi is 62. (3) Because the type of this item is C, *Length* should also be parsed. The bit part of *Length* is "10", so the *Length* is 1. The decoding of the first item is complete. After parsing all the items in the same way, Table 7 is created and the remaining data R_{stage_2} is as follows.

$$
\begin{aligned}
R_{stage_2} = \; &110010001110100010111011110010001101111\\
&1111010110000011111010001001100110011000\\
&1110011101000111111000100110
\end{aligned}
$$

3) DECODING CONCATENATED CHAIN CODES

After the previous stage, the remaining encoded parts are the concatenated chain codes. In the decoding process of the encoded parts, we first create a bitstream that is the same size as the image, initialized to zero, and analyze the items in the transformed chain code table one by one to change part of the bitstream to one. Depending on the decoded item's type, the concatenated chain codes are also decoded using Table 8. A pseudo algorithm of the decoding process is shown in algorithm 3. For example, the Δi of the first item in TT_{bk} is 62, so $B_{bk}[61]$ is set to 1. Since the type of this item is C and *Length* is 1, one code is decoded from R_{stage_2} using Table 8. And the result is 1, so $B_{bk}[87]$ is set to 1. As a result of this stage, B_{bk} and B_{rd} are obtained as shown in Section IV-B1.

4) REVERSE BINARY MTFT

This stage is the final of the decompression process. Reverse Binary MTFT is performed on the bitstream generated from the previous stage. A pseudo algorithm of the transform process is shown in algorithm 4. It could be improved to work faster like algorithm 5, which uses more memory. B_{bk} and B_{rd} are converted to O_{bk} and O_{rd} in this stage as shown in Figure 13.

These represent images, as shown in Figure 14. These bitstreams may be passed on to the e-paper module and used but may require additional conversion processes. This process depends on the manufacturer of the module and is not covered in this paper.

V. PERFORMANCE EVALUATION

In this section, we evaluated the performance of the proposed algorithm with six well-known compression algorithms, such as PackBits, LZW, Deflate, G3, G4, and JBIG2.

TABLE 9. The size and color composition information of the images.

ID	Format	File Size	Width	Height	Black Pixels	Red Pixels	Blue Pixels	Orange Pixels	Magenta Pixels	Green Pixels	Yellow Pixels	White Pixels
Label01	PBM	23115	152	152	5531	-	-	-	-	-	-	17573
Label02	PBM	23115	152	152	8073	-	-	-	-	-	-	15031
Label03	PBM	23115	152	152	12494	-	-	-	-	-	-	10610
Label04	PBM	23115	152	152	14674	-	-	-	-	-	-	8430
Label05	PBM	32779	250	128	6533	-	-	-	-	-	-	25467
Label06	PBM	32779	250	128	13152	-	-	-	-	-	-	18848
Label07	PBM	32779	250	128	19242	-	-	-	-	-	-	12758
Label08	PBM	32779	250	128	23649	-	-	-	-	-	-	8351
Label09	PBM	120011	400	300	24454	-	-	-	-	-	-	95546
Label10	PBM	120011	400	300	58921	-	-	-	-	-	-	61079
Label11	PBM	120011	400	300	74136	-	-	-	-	-	-	45864
Label12	PBM	120011	400	300	87614	-	-	-	-	-	-	32386
Label13	PPM	69327	152	152	3985	1546	-	-	-	-	-	17573
Label14	PPM	69327	152	152	3623	4449	-	-	-	-	-	15032
Label15	PPM	69327	152	152	2112	10382	-	-	-	-	-	10610
Label16	PPM	69327	152	152	1865	12809	-	-	-	-	-	8430
Label17	PPM	96015	250	128	4164	2369	-	-	-	-	-	25467
Label18	PPM	96015	250	128	4201	8951	-	-	-	-	-	18848
Label19	PPM	96015	250	128	2383	16859	-	-	-	-	-	12758
Label20	PPM	96015	250	128	1149	22500	-	-	-	-	-	8351
Label21	PPM	360015	400	300	11830	12624	-	-	-	-	-	95546
Label22	PPM	360015	400	300	16277	42644	-	-	-	-	-	61079
Label23	PPM	360015	400	300	6579	67557	-	-	-	-	-	45864
Label24	PPM	360015	400	300	5525	82089	-	-	-	-	-	32386
Label25	PPM	69327	152	152	1632	1546	1226	4398	520	607	6510	6665
Label26	PPM	69327	152	152	2325	2477	2843	258	169	5275	4753	5004
Label27	PPM	69327	152	152	963	9488	4323	1730	313	1580	1512	3195
Label28	PPM	69327	152	152	1865	4114	5419	544	282	1750	2994	6136
Label29	PPM	96015	250	128	2285	4871	2349	12631	520	1379	5048	2917
Label30	PPM	96015	250	128	3073	3582	1128	5249	3207	2162	7773	5826
Label31	PPM	96015	250	128	1234	1462	13865	1885	2681	1542	5101	4230
Label32	PPM	96015	250	128	850	10100	7740	4660	299	780	1946	5625
Label33	PPM	360015	400	300	10236	1842	12981	47724	18518	8558	10782	9359
Label34	PPM	360015	400	300	16277	18401	19401	2564	4842	18190	27386	12939
Label35	PPM	360015	400	300	6579	30611	10545	13382	7391	16173	15957	19362
Label36	PPM	360015	400	300	4878	1702	4717	32158	12361	14832	21683	27669

Three performance items were measured: compressed file size, compression time, and decompression time.

Figure 15 shows thirty-six images used for the evaluation. The images are classified into three categories by the number of colors: monochrome, 3-colors, and 8-colors. Note that due to ESL tags' limited resource constraints, only limited colored tags have been deployed in the markets. The size and color composition information of the images is shown in Table 9.

In the performance evaluation, we converted multi-colored images into multiple monochrome images because G3, G4, and JBIG2 can compress only monochrome images. For PackBits, LZW, and Deflate, we compared two different versions of images: one multi-colored image (A) and multiple converted monochrome images (B). For instance, PackBits (A) refers to the performance result for the original multi-colored image, while PackBits (B) refers to the result for the converted monochrome images.

For the evaluation, we used the source codes of all the compression algorithms except ECO from JBIG2 [27], [28] and LibTiff [29]. The performance of all algorithms was measured in the same environment.[1] All measurements were repeated 1000 times, and the average value was used for the comparison.

Table 10 shows the comparison of compressed image sizes with each algorithm. For the monochrome images, ECO and JBIG2 showed the lowest BPP (bits-per-pixel), indicating the highest compression ratio. However, for the 3-color and 8-color images, ECO showed the highest compression ratio. It can be seen that the larger the color palette's size, the better the ECO's compression efficiency than the other algorithms.

Table 11 shows the comparison of the compression time for each algorithm. Figure 17 shows the average compression time for each group of the images. For the monochrome images, ECO and JBIG2 showed longer compression time than the other algorithms. However, for the 3-color and 8-color images, ECO showed shorter compression time than JBIG2.

Table 12 shows the comparison of the decompression time for each algorithm. Figure 18 shows the average decompression time for each group of the images. For the monochrome and 3-color images, ECO showed the shortest decompression time. For the 8-color images, PackBits(A), LZW(A), and Deflate(A) showed shorter decompression times than ECO; Note that they showed relatively lower compression ratio than ECO as shown in Table 10.

[1] Hardware: Intel(R) Core(TM) i7-5650U CPU @ 2.20GHz; 2 Cores, 4 logical Processors; 8 GB RAM Software: Xcode version 11.2.1, macOS 10.15 (Catalina)

TABLE 10. Compressed size of the images (in bytes).

ID	Uncompressed	PackBits(A)	PackBits(B)	LZW(A)	LZW(B)	Deflate(A)	Deflate(B)	G3	G4	JBIG2	ECO
Label01	23115	2092	-	1390	-	1064	-	2094	1092	700	784
Label02	23115	2320	-	1464	-	1110	-	2226	1070	812	806
Label03	23115	1992	-	1330	-	972	-	2020	944	742	711
Label04	23115	2336	-	1506	-	1058	-	2404	1126	876	791
Label05	32779	2342	-	1506	-	1108	-	2344	1174	718	806
Label06	32779	3104	-	1776	-	1270	-	2682	1194	907	857
Label07	32779	2508	-	1512	-	1012	-	2248	970	726	688
Label08	32779	2810	-	1688	-	1138	-	2420	1112	910	830
Label09	120011	7336	-	3960	-	2552	-	5494	2116	1317	1588
Label10	120011	8428	-	4316	-	2920	-	6148	2194	1604	1692
Label11	120011	7384	-	4010	-	2510	-	5882	2108	1540	1583
Label12	120011	7966	-	4286	-	2668	-	6494	2328	1692	1688
Label13	69327	11446	4172	2154	2542	1608	1258	2710	1260	831	787
Label14	69327	19760	4592	2328	2840	1724	1312	2866	1242	940	820
Label15	69327	36154	4486	2094	2494	1582	1186	2650	1114	879	717
Label16	69327	43910	5238	2246	2904	1768	1268	3046	1296	998	812
Label17	96015	14876	5244	2600	2744	1740	1320	2920	1338	868	815
Label18	96015	34484	6336	3128	3584	2062	1488	3276	1360	1039	876
Label19	96015	56388	5812	2460	2964	1764	1230	2840	1136	858	694
Label20	96015	72240	6232	2784	3282	1954	1334	3016	1280	1041	839
Label21	360015	55288	16208	7822	8182	4274	2776	6748	2306	1486	1593
Label22	360015	143550	17882	8642	9456	4816	3194	7424	2396	1769	1707
Label23	360015	217038	18700	8208	8382	4462	2746	7094	2300	1729	1603
Label24	360015	260840	20368	9032	8996	4684	2916	7748	2522	1841	1671
Label25	69327	49204	4940	2136	5690	3360	2616	6460	2430	1851	1124
Label26	69327	51902	5076	2216	6050	3510	2676	6606	2412	1961	1167
Label27	69327	60176	5082	2078	6098	3682	2792	6608	2490	2117	1290
Label28	69327	50204	5488	2320	5800	3324	2516	6440	2332	1914	1039
Label29	96015	85328	6668	2756	6510	4186	3102	7072	2878	2252	1511
Label30	96015	75576	7208	3070	7208	4234	3058	7038	2558	2115	1285
Label31	96015	83796	6636	2412	6532	3922	2794	6536	2420	2042	1187
Label32	96015	80458	6710	2764	5936	3682	2642	6294	2386	1951	1129
Label33	360015	313382	20954	7718	17126	8872	5686	15740	4388	3441	2659
Label34	360015	284428	20782	8666	18104	9114	5622	15314	4044	3211	2445
Label35	360015	294152	20976	8880	15134	7914	4582	14228	3756	3117	2087
Label36	360015	275372	21070	9084	14178	7368	4558	14300	3842	2942	2024

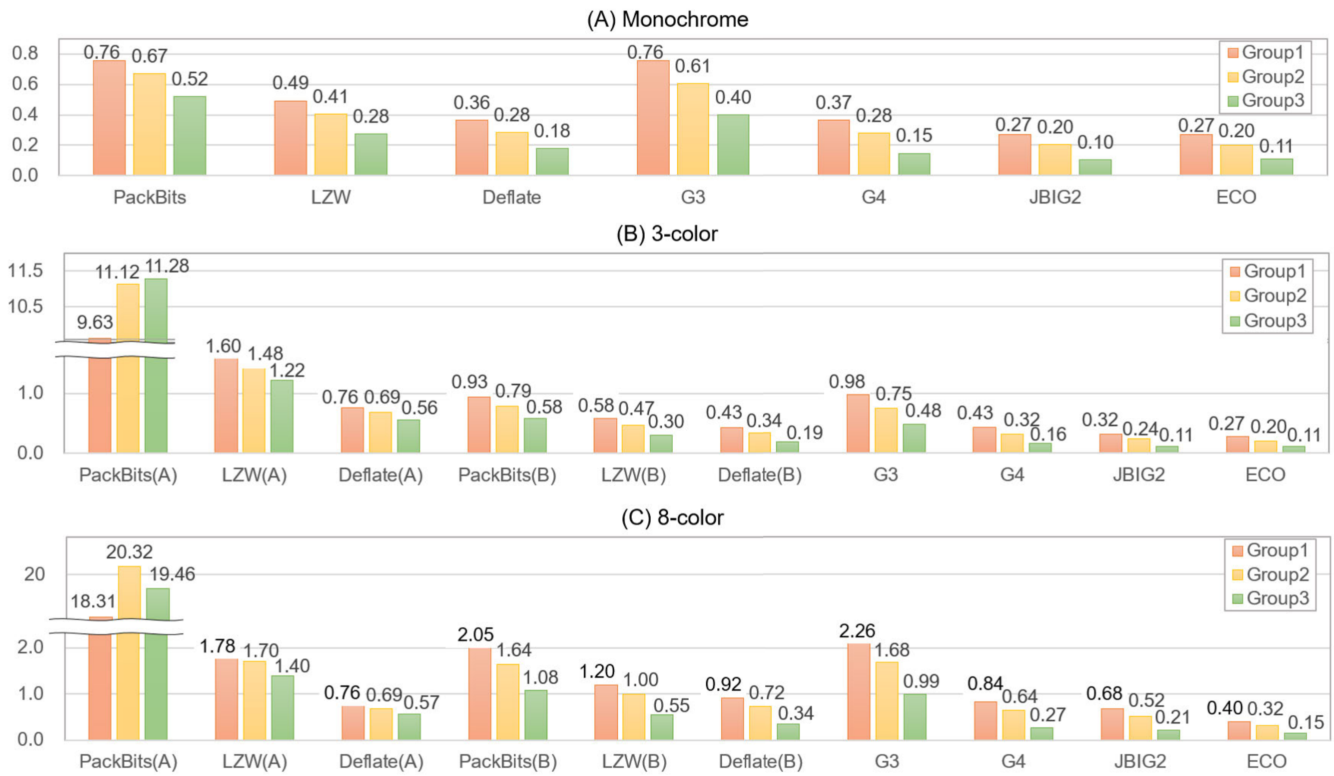


FIGURE 16. Average bits-per-pixel comparison for each group.

TABLE 11. Compression time comparison (in milli-seconds).

ID	PackBits(A)	PackBits(B)	LZW(A)	LZW(B)	Deflate(A)	Deflate(B)	G3	G4	JBIG2	ECO
Label01	0.2845	-	0.4038	-	0.5893	-	0.3694	0.4466	1.4975	1.1274
Label02	0.2860	-	0.4048	-	0.5860	-	0.3718	0.4506	1.5238	1.1355
Label03	0.2824	-	0.4004	-	0.5847	-	0.3658	0.4372	1.5306	1.0972
Label04	0.2855	-	0.4064	-	0.5625	-	0.3762	0.4562	1.5255	1.1704
Label05	0.2871	-	0.4071	-	0.5915	-	0.3816	0.4720	1.5991	1.3138
Label06	0.2915	-	0.4159	-	0.6165	-	0.3887	0.4868	1.7176	1.3681
Label07	0.2859	-	0.4107	-	0.5747	-	0.3751	0.4503	1.5835	1.2667
Label08	0.2899	-	0.4146	-	0.5843	-	0.3808	0.4679	1.5758	1.3482
Label09	0.3363	-	0.5491	-	0.8627	-	0.4922	0.6642	3.5688	3.4804
Label10	0.3452	-	0.5458	-	1.0323	-	0.4995	0.6783	3.2219	3.6091
Label11	0.3365	-	0.5253	-	0.7949	-	0.4917	0.6656	3.5396	3.6276
Label12	0.3386	-	0.5393	-	0.8175	-	0.5071	0.6931	3.3415	3.6991
Label13	0.4123	0.7644	1.3957	1.3560	1.5740	1.8704	1.4816	1.5558	2.8553	1.2602
Label14	0.4547	0.7688	1.5375	1.3612	1.5776	1.8828	1.4891	1.5652	2.8829	1.2495
Label15	0.5195	0.7681	1.4388	1.3737	1.5913	1.8929	1.5017	1.5690	2.8779	1.2064
Label16	0.5496	0.7798	1.5375	1.3890	1.6070	1.8876	1.5235	1.6068	2.9079	1.2971
Label17	0.4536	0.9037	1.6031	1.4854	1.7107	2.0024	1.6223	1.7164	3.1334	1.4859
Label18	0.5431	0.9274	2.0032	1.5085	1.7361	2.0553	1.6495	1.7427	3.2351	1.5309
Label19	0.6303	0.9205	1.6310	1.5214	1.7458	2.0212	1.6444	1.7253	3.1773	1.4314
Label20	0.6963	0.9220	1.7931	1.5413	1.7646	2.0417	1.6716	1.7654	3.1992	1.5264
Label21	0.9255	2.3132	4.1030	2.7744	3.0943	3.5427	2.9684	3.1498	6.9584	4.1102
Label22	1.2731	2.3469	4.8619	2.8883	3.2121	3.7658	3.0805	3.2706	6.5890	4.2031
Label23	1.5539	2.3627	4.4959	2.9151	3.2433	3.6381	3.1110	3.2982	7.0799	4.2388
Label24	1.7276	2.3991	4.9748	2.9809	3.3014	3.7185	3.1895	3.3811	6.9604	4.3517
Label25	0.5532	0.7710	1.4592	3.2107	3.9591	4.8510	3.5572	3.6717	6.6183	2.0544
Label26	0.5617	0.7655	1.5359	3.1359	3.8796	4.7850	3.4806	3.5999	6.5851	2.0645
Label27	0.5920	0.7665	1.3947	3.1812	3.9283	4.8346	3.5324	3.6530	6.7536	2.1061
Label28	0.5600	0.7663	1.5398	3.1286	3.8948	4.7173	3.4939	3.5944	6.5621	2.0284
Label29	0.7040	0.9196	1.7452	3.3933	4.1719	5.0867	3.7684	3.9348	7.6741	2.7520
Label30	0.6935	0.9132	1.9561	3.4094	4.1843	5.0985	3.7676	3.9163	7.4146	2.6077
Label31	0.7108	0.9051	1.5765	3.4078	4.1881	5.0388	3.7452	3.8887	7.4021	2.5665
Label32	0.7171	0.9114	1.7046	3.4369	4.2215	5.0303	3.7870	3.9169	7.4040	2.5338
Label33	1.8541	2.3941	4.1273	5.3793	6.4213	7.7056	5.8316	6.1441	15.8971	8.2895
Label34	1.8046	2.3767	4.6854	5.3695	6.4238	7.8205	5.8065	6.0718	15.1071	8.1834
Label35	1.8295	2.4042	4.9235	5.3126	6.3811	7.4063	5.7526	6.0032	15.4273	7.9932
Label36	1.7101	2.4199	4.8496	5.2923	6.3393	7.3886	5.7356	5.9826	14.6490	8.0444

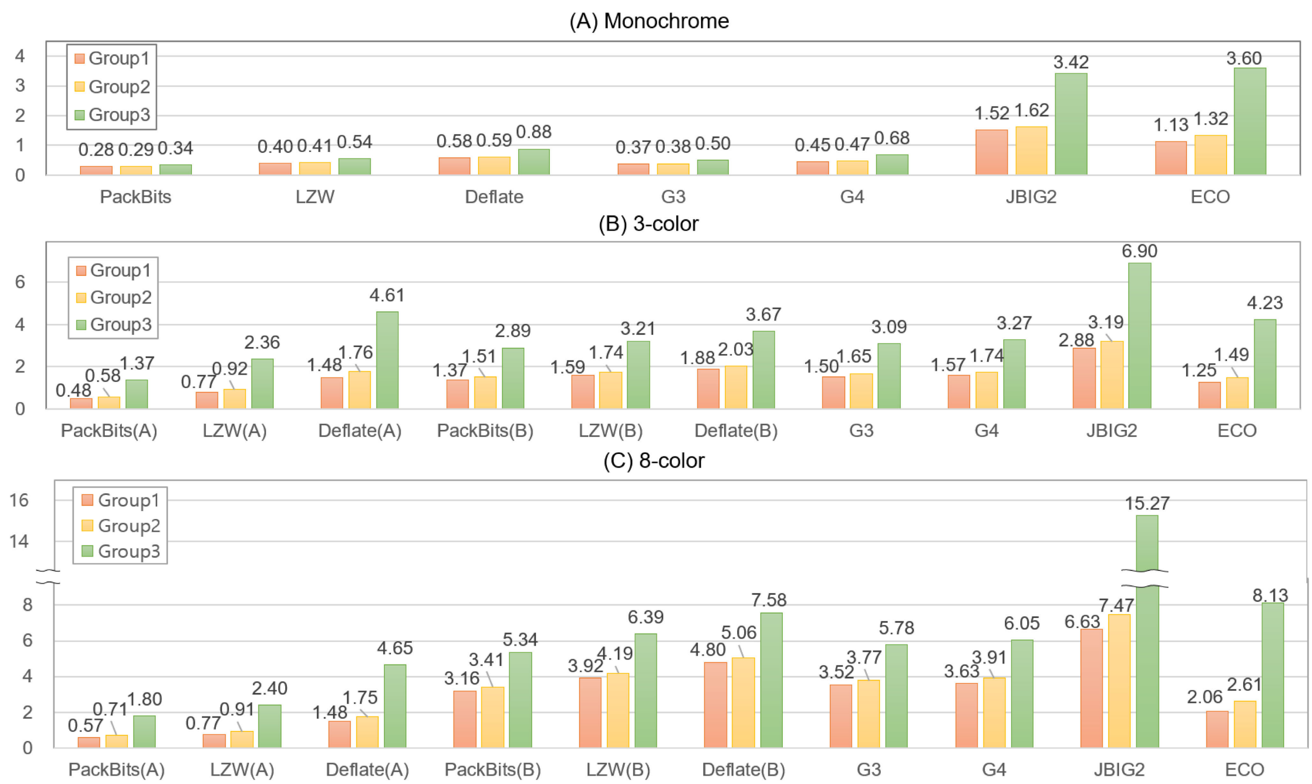


FIGURE 17. Average compression time comparison for each group (in milli-seconds).

TABLE 12. Decompression time comparison (in milli-seconds).

ID	PackBits(A)	PackBits(B)	LZW(A)	LZW(B)	Deflate(A)	Deflate(B)	G3	G4	JBIG2	ECO
Label01	0.5303	-	0.6076	-	0.5774	-	0.6441	0.6444	0.4514	0.3478
Label02	0.5274	-	0.6118	-	0.5768	-	0.6395	0.6493	0.5020	0.3420
Label03	0.5275	-	0.6117	-	0.5712	-	0.6383	0.6382	0.5204	0.3201
Label04	0.5337	-	0.6102	-	0.5709	-	0.6420	0.6493	0.5288	0.3488
Label05	0.5772	-	0.6576	-	0.6236	-	0.6930	0.6940	0.4932	0.3608
Label06	0.5729	-	0.6537	-	0.6246	-	0.6950	0.6959	0.5958	0.3702
Label07	0.5751	-	0.6509	-	0.6166	-	0.6862	0.6853	0.5752	0.3209
Label08	0.5759	-	0.6649	-	0.6144	-	0.6859	0.6987	0.5969	0.3563
Label09	1.0057	-	1.1298	-	1.0336	-	1.1753	1.2010	1.2684	0.6017
Label10	1.0300	-	1.1227	-	1.0454	-	1.1875	1.1628	1.2063	0.6133
Label11	1.0097	-	1.3228	-	1.0297	-	1.1480	1.1204	1.3407	0.6035
Label12	0.9788	-	1.0731	-	1.0360	-	1.1622	1.1383	1.2318	0.6297
Label13	0.5815	0.7583	0.6649	1.0607	1.1997	1.1405	1.2256	1.2167	0.7289	0.4523
Label14	0.5862	0.7566	0.6583	1.0594	1.1981	1.1671	1.2396	1.2254	0.7488	0.4472
Label15	0.5907	0.7565	0.6530	1.0563	1.2023	1.1461	1.2363	1.2140	0.8035	0.4187
Label16	0.6066	0.7626	0.6449	1.0600	1.2003	1.1440	1.2384	1.2252	0.8067	0.4527
Label17	0.6389	0.8606	0.7572	1.1528	1.2874	1.2416	1.3334	1.3034	0.7810	0.4679
Label18	0.6494	0.8688	0.7434	1.1587	1.2841	1.2365	1.3452	1.3138	0.8144	0.4784
Label19	0.6581	0.8753	0.7354	1.1598	1.2855	1.2308	1.3336	1.2987	0.8575	0.4302
Label20	0.6651	0.8632	0.7216	1.1395	1.2829	1.2326	1.3343	1.2996	0.8789	0.4691
Label21	1.2001	1.8646	1.5809	2.0185	2.1837	2.1602	2.2903	2.1599	1.5885	0.7827
Label22	1.2467	1.8738	1.4933	2.0430	2.1568	2.1669	2.2922	2.1852	1.4244	0.7914
Label23	1.2603	1.9137	1.4661	1.9981	2.1567	2.1247	2.2474	2.1918	1.6614	0.7923
Label24	1.3051	1.8963	1.4461	2.0368	2.1387	2.1288	2.2591	2.2472	1.5463	0.8099
Label25	0.5811	0.7617	0.6446	3.6469	4.1928	3.9802	4.1616	4.1362	2.2697	0.9362
Label26	0.5817	0.7628	0.6426	3.6672	4.1544	3.9676	4.1695	4.1311	2.2808	0.9316
Label27	0.5837	0.7675	0.6450	3.6833	4.1742	3.9778	4.1493	4.1383	2.3205	0.9593
Label28	0.5918	0.7633	0.6441	3.6676	4.1668	3.9510	4.1629	4.1330	2.2914	0.9134
Label29	0.6505	0.8670	0.7255	4.0034	4.4856	4.3064	4.4869	4.4608	2.5018	1.0704
Label30	0.6557	0.8736	0.7191	3.9730	4.5042	4.3069	4.4674	4.4681	2.4379	0.9962
Label31	0.6500	0.8704	0.7260	3.9079	4.4981	4.3140	4.4772	4.3952	2.4163	0.9836
Label32	0.6512	0.8737	0.7201	3.9579	4.4909	4.3150	4.4624	4.3913	2.4137	0.9680
Label33	1.2416	1.9364	1.4506	6.9010	7.5138	7.2049	7.5947	7.4953	3.9541	1.8057
Label34	1.2699	1.9272	1.4915	6.9437	7.5828	7.2256	7.5421	7.5124	3.6873	1.7375
Label35	1.2729	1.9471	1.4784	6.9453	7.5737	7.2674	7.5414	7.4634	3.9580	1.6650
Label36	1.2573	1.9593	1.5004	6.8711	7.4480	7.2567	7.5106	7.4573	3.1668	1.7002

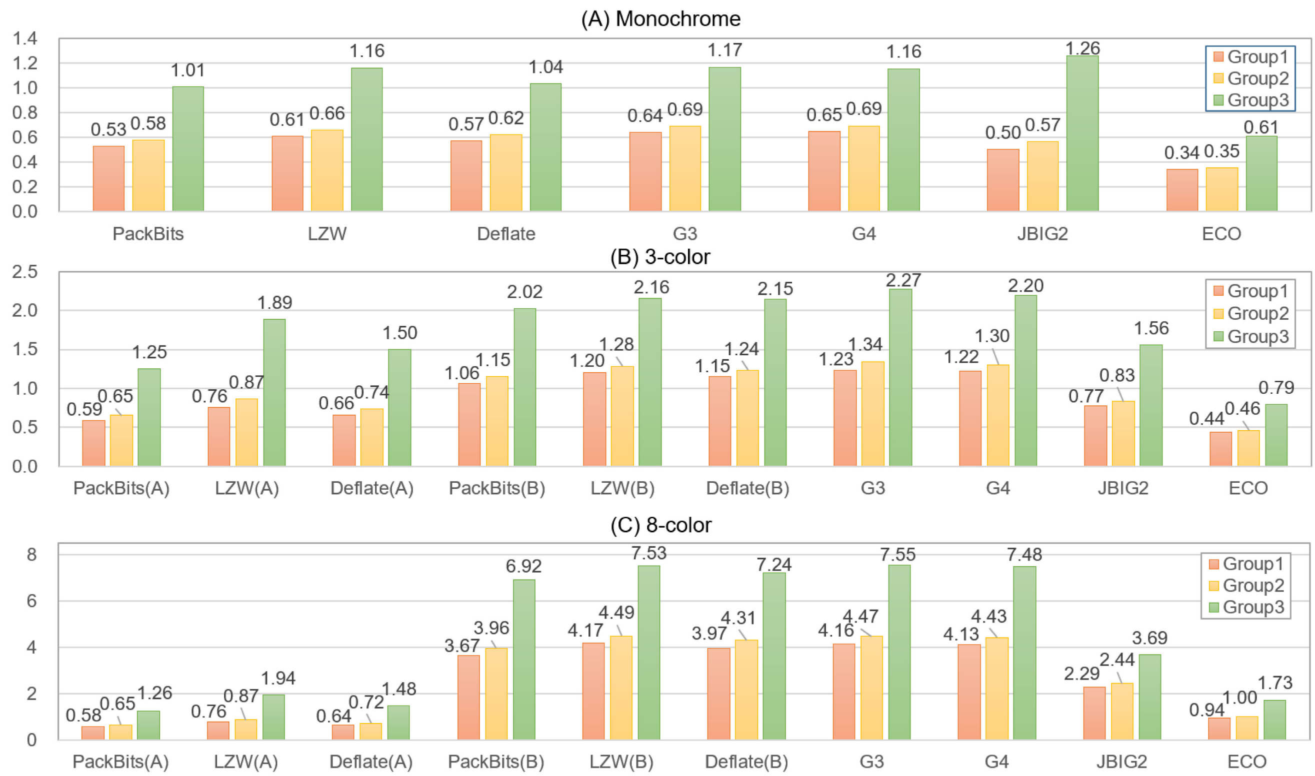


FIGURE 18. Average decompression time comparison for each group (in milli-seconds).

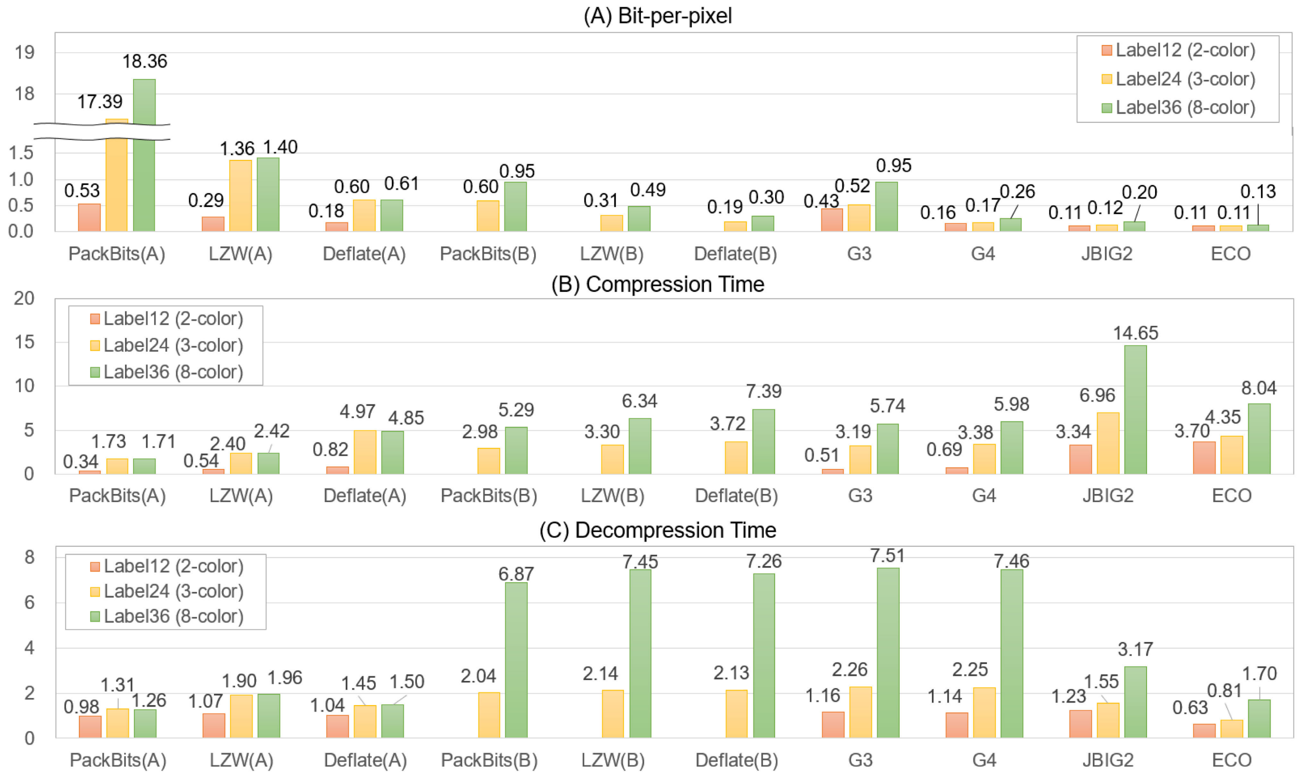


FIGURE 19. Performance results of the algorithms for the images with Label12, 24, and 36.

TABLE 13. Compression ratio comparison of the algorithms.

		Values	ECO	PackBits(A)	PackBits(B)	LZW(A)	LZW(B)	Deflate(A)	Deflate(B)	G3	G4	JBIG2
Monochrome	Group 1	CR	29.97	10.63	-	16.29	-	22.04	-	10.62	21.95	29.76
		CR/CR_{ECO}	1.00	0.35	-	0.54	-	0.74	-	0.35	0.73	0.99
	Group 2	CR	41.51	12.32	-	20.33	-	29.15	-	13.58	29.66	40.74
		CR/CR_{ECO}	1.00	0.30	-	0.49	-	0.70	-	0.33	0.71	0.98
	Group 3	CR	73.57	15.48	-	29.01	-	45.23	-	20.06	54.97	78.70
		CR/CR_{ECO}	1.00	0.21	-	0.39	-	0.61	-	0.27	0.75	1.07
3-Color	Group 1	CR	88.68	25.84	3.27	41.59	15.10	55.27	31.48	24.67	56.64	76.38
		CR/CR_{ECO}	1.00	0.29	0.04	0.47	0.17	0.62	0.36	0.28	0.64	0.86
	Group 2	CR	120.05	30.86	3.07	51.33	16.35	71.83	35.29	31.96	75.47	101.79
		CR/CR_{ECO}	1.00	0.26	0.03	0.43	0.14	0.60	0.29	0.27	0.63	0.85
	Group 3	CR	219.24	41.26	3.01	79.13	19.82	124.24	42.85	49.76	151.41	212.39
		CR/CR_{ECO}	1.00	0.19	0.01	0.36	0.09	0.57	0.20	0.23	0.69	0.97
8-Color	Group 1	CR	60.39	11.74	1.32	20.02	13.49	26.20	31.75	10.62	28.71	35.44
		CR/CR_{ECO}	1.00	0.19	0.02	0.33	0.22	0.43	0.53	0.18	0.48	0.59
	Group 2	CR	76.05	14.74	1.18	24.04	14.12	33.26	35.16	14.29	37.70	46.07
		CR/CR_{ECO}	1.00	0.19	0.02	0.32	0.19	0.44	0.46	0.19	0.50	0.61
	Group 3	CR	158.25	22.52	1.24	43.61	17.19	71.23	42.09	24.22	90.16	113.65
		CR/CR_{ECO}	1.00	0.14	0.01	0.28	0.11	0.45	0.27	0.15	0.57	0.72

In Figure 19, we compared the algorithms on the images with the same contents but different color-palettes. For instance, Label12, Label24, and Label36 have the same characteristics, such as size, structure, and contents, but different colors. Overall, as the number of colors increases, all of the algorithms showed the lower compression ratio and took more time in compression and decompression. Notice that ECO showed relatively lower performance variations for the color-palette change.

Table 13 shows the comparison of the compression ratio CR of each algorithm which is defined by Equation 5. It also shows CR/CR_{ECO}, a relative compression ratio to ECO. For instance, if it is greater than 1, it indicates that the algorithm generates smaller compressed image than ECO. Notice that the relative compression ratios are less than 1 in most cases in Table 13.

$$CR = \frac{uncompressed\ size}{compressed\ size} \tag{5}$$

VI. CONCLUSION

An Electronic shelf labels (ESL) system is becoming an attractive alternative for managing up-to-date price tag information because of the dynamic-price-updating and customer's product-evaluation-display features. A common ESL system configuration in a retail store includes thousands of battery-powered ESL tags that are mostly connected wirelessly in a dense indoor environment. Raising the success ratio of wireless communication is essential for the system's viability due to its limited battery life. In this paper, we presented ECO, a new chain coding based image compression algorithm suitable for the ESL system. We evaluated the performance of ECO against six other well-known algorithms. ECO showed the best results in the compression ratio and the decompression time in most cases. Achieving high compression ratio and short decompression time together is one of the most important factors for prolonging the battery lifetime of ESL tags.

REFERENCES

[1] H. A. Al-Kashoash, F. Hassen, H. Kharrufa, and A. H. Kemp, "Analytical modeling of congestion for 6LoWPAN networks," *ICT Exp.*, vol. 4, no. 4, pp. 209–215, 2018.

[2] N. Sahoo, C. Dellarocas, and S. Srinivasan, "The impact of online product reviews on product returns," *Inf. Syst. Res.*, vol. 29, no. 3, pp. 723–738, Sep. 2018.

[3] J. Xu and W. Li, "Design of electronic shelf label based on electronic paper display," in *Proc. 3rd Int. Conf. Consum. Electron., Commun. Netw.*, Nov. 2013, pp. 250–253.

[4] D. Rohm, M. Goyal, H. Hosseini, A. Divjak, and Y. Bashir, "A simulation based analysis of the impact of IEEE 802.15.4 MAC parameters on the performance under different traffic loads," *Mobile Inf. Syst.*, vol. 5, no. 1, pp. 81–99, 2009.

[5] M. A. Rahman and M. Hamada, "Lossless image compression techniques: A state-of-the-art survey," *Symmetry*, vol. 11, no. 10, p. 1274, Oct. 2019.

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

[7] K. Khursheed, M. Imran, N. Ahmad, and M. O'Nils, "Selection of bi-level image compression method for reduction of communication energy in wireless visual sensor networks," *Proc. SPIE*, vol. 8437, May 2012, Art. no. 84370M.

[8] J. Sung, "End of paper labels: Emerging smart labels toward Internet of Things," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 216–221.

[9] *IEEE Standard for Local and Metropolitan Area Networks—Part 16: Air Interface for Fixed Broadband Wireless Access Systems*, IEEE Standard 802.16-2004, IEEE 802.16 Working Group, 2004.

[10] C. H. Zhou, P. Mei, L. W. Huang, K. Z. Liu, and Y. Q. Wen, "An electronic shelf label system based on WSN," *Adv. Mater. Res.*, vol. 765, pp. 1718–1721, Sep. 2013.

[11] J.-S. Park and B.-J. Jang, "Electronic shelf label system employing a visible light identification link," in *Proc. IEEE Radio Wireless Symp. (RWS)*, Jan. 2016, pp. 219–222.

[12] J. Ock, H. Kim, H.-S. Kim, J. Paek, and S. Bahk, "Low-power wireless with denseness: The case of an electronic shelf labeling system—Design and experience," *IEEE Access*, vol. 7, pp. 163887–163897, 2019.

[13] C. E. Sannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, 1948.

[14] B. Y. Ryabko, "Data compression by means of a 'book stack,"' *Problemy Peredachi Informatsii*, vol. 16, no. 4, pp. 16–21, 1980.

[15] B. Y. Ryabko, "Comments to: 'A locally adaptive data compression scheme' by JL Bentley, DD Sleator, RE Tarjan, and VK Wei," *Commun. ACM*, vol. 30, no. 9, pp. 792–796, 1987.

[16] H. Freeman, "On the encoding of arbitrary geometric configurations," *IRE Trans. Electron. Comput.*, vol. 2, pp. 260–268, Jun. 1961.

[17] B. Žalik and N. Lukač, "Chain code lossless compression using move-to-front transform and adaptive run-length encoding," *Signal Process., Image Commun.*, vol. 29, no. 1, pp. 96–106, Jan. 2014.

[18] B. Žalik, D. Mongus, K. R. Žalik, and N. Lukač, "Chain code compression using string transformation techniques," *Digit. Signal Process.*, vol. 53, pp. 1–10, Jun. 2016.

[19] E. Bribiesca, "A new chain code," *Pattern Recognit.*, vol. 32, no. 2, pp. 235–251, 1999.

[20] H. Sanchez-Cruz and R. M. Rodrguez-Dagnino, "Compressing bi-level images by means of a 3-bit chain code," *Opt. Eng.*, vol. 44, no. 9, pp. 1–8, 2005.

[21] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984.

[22] P. Deutsch, *DEFLATE Compressed Data Format Specification Version 1.3*, document RFC 1951, 1996.

[23] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[24] *JBIG2 Bi-Level Image Compression Standard*, Standard ISO/IEC 14492, ITU-T Rec. T.88, 2000.

[25] *Standardization of Group 3 Facsimile Terminals for Document Transmission*, document ITU-T Rec. T.4, 2003.

[26] *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, document ITU-T Rec. T.6, 1988.

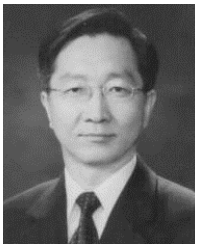
[27] A. Langley. (2019). JBIG2 encoder. GitHub. [Online]. Available: https://github.com/agl/jbig2enc

[28] R. Giles. (2019). JBIG2 decoder. GitHub. [Online]. Available: https://github.com/rillian/jbig2dec

[29] (2019). *LibTiff—TIFF Library and Utilities*. [Online]. Available: https://http://www.libtiff.org

YOUNGJUN KIM received the B.S. degree in information and computer engineering from Ajou University, in 2010, where he is currently pursuing the Ph.D. degree with the Department of Computer Engineering. His research interests include designing protocols and techniques for the massive Internet of Things (IoT).

KI-HYUNG KIM (Senior Member, IEEE) received the M.S. and Ph.D. degrees in electrical and electronic engineering from the Korea Advanced Institute of Science and Technology (KAIST), in 1990 and 1996, respectively. He is currently a Professor with the Department of Cyber Security, College of Information Technology. His research interests include blockchain, cybersecurity, the IoT, and embedded systems.



WE-DUKE CHO received the M.S. and Ph.D. degrees in electrical and electronic engineering from the Korea Advanced Institute of Science and Technology (KAIST), in 1983 and 1987, respectively. He is currently a Professor with the Department of Electronics Engineering, College of Information Technology. He is also the Director of the Ubiquitous Convergence Research Institute (UCRI). He is also with the Life-Care Science Laboratory, Ajou University, South Korea. He is also the Director of the Korea Association of Smart Home (KASH). He is also an Adjunct Professor with Stony Brook University (SUNY), Stony Brook, NY, USA. He presented a talk entitled "A design for lifestyle analyzer based on IoT Big data with Human Behavior Recognition". His research interests include signal processing, the IoT data pattern modeling, and care system design.