A Modified Version of Huffman Coding with Random Access Abilities

Ke Yang, Sian-Jheng Lin*, Honggang Hu
University of Science and Technology of China, China
email: yk303943@mail.ustc.edu.cn, sjlin@ustc.edu.cn, hghu2005@ustc.edu.cn

***Abstract*—In this paper, a modified version of canonical Huffman coding is presented, such that a certain level of random access on the compressed file is allowed. Compared with the prior methods, the proposed approach does not need additional space to store the auxiliary information. Thus, it is possible to provide a certain level of random access ability on a compressed file whose size is close to the Huffman coding asymptotically. To our knowledge, this is the first method to possess random access ability without the help by the auxiliary data. Though the Huffman coding is discussed in this paper, the present method can be applied on any variable-length prefix encoding.**

Keywords*—Huffman coding; random access; neighbor-based storage

I. Introduction

Data compression is a technique for reducing the storage space required for a given piece of information [1]. Entropy coding is a family of lossless compression techniques for the i.i.d. source sequences. Huffman coding [2] is one of the most representative entropy coding technologies.

Huffman coding is based on the source content of each symbol occurrence probability of variable-length code, probability can be based on a prior statistics or the probabilities in the source sequence [3], [4]. The code is called dynamic Huffman coding when the estimated probabilities are updated during the encoding [5], or else the code is called static Huffman coding. In this paper, a subclass of static Huffman coding, canonical Huffman code is considered.

Random access is a property that any data element can be accessed from a set of addressable elements efficiently. In some applications, the random access on encoded files is a desired ability. For example, [6] presents a full-text retrieval system based on this technology. Moreover, this ability allows that users can access any items in a compressed table within a database without uncompressing the whole table.

For the source sequence (i.e. the uncompressed file), the symbols are encoded by equal-length coding, and hence it is easy to access any symbols by their indices. However, for the file encoded by variable-length coding, the size of each symbol would be different, and thus it is more difficult to identify the location of a particular symbol in the compressed file. In this case, we have to decompress the file sequentially, until obtaining the desired symbol. Obviously, this approach is expensive because the cost of accessing a symbol is proportional to its index. To solve this issue, some approaches are proposed. For example, one can separate the file into many non-overlapping blocks, and then build an indexed file that stores the bit-addresses of each block [7]. In [8], a neighbor-based block-organized storage scheme is introduced to support local accesses. The scheme stores the compressed sequences with different lengths within blocks of fixed size according to the "neighbor-based" rule. However, these two methods cannot achieve the compression rate that Huffman coding can reach. In particular, [7] needs additional storage space to store the locations of the start points of some symbols, and the compression technoque in [8] is the length-prefix compression, whose compression ratio is worse than Huffman coding. Notably, if the source sequence is a monotone sequence, [9] introduces the Elias-Fano representation for it. The Elias-Fano encoding scheme partition the high bits into buckets by strictly monotonous ordering and join the lowest ℓ bits together. In this paper, we present a compression method, and its compression ratio approaches that of Huffman coding. Further, a certain level of random access is supported on the compressed file. The proposed compression method applys the neighbor-based rule [8] on the canonical Huffman coding. Precisely, the method rearranges the bit ordering of the Huffman codewords by the neighbor-based rule. This allows us to supports a certain level of random access on the compressed file without auxiliary information (except for some parameters). To our knowledge, is the first statistic symbol-by-symbol entropy encoding to approach the entropy limit with preserving the random access abilities.Contributions of this paper are presented as follows:

1) This paper presents an entropy coding scheme, which possesses the random access ability without the help by the auxiliary data.
2) We suggest to adopt the canonical Huffman code in the proposed scheme to reduce the number of accessed bits in the symbol-retrieving process.

The rest of this paper is organized as follows. In Section II, we introduce the canonical Huffman code, the neighbor-based storage scheme, and some notations used in this paper. Section III presents the encoding algorithm, and Section IV presents the method to access the k-th symbol, for $k \in \{1, 2, 3, ..., N\}$. Section V presents the decoding algorithm. In Section VI, we analysis the storage efficiency and the cost of retrieving a symbol. Section VII concludes the work.

978-1-5386-8339-2/18/$31.00 ©2018 IEEE

II. Preliminaries

A. Notations

The following lists the notations used in this paper.

1) For a sequence A, $|A|$ denotes length of A, and the i-th symbol of A is denoted by $A[i]$. Further, $A[i:j] := (A[i], A[i+1], \ldots, A[j])$ denotes a subsequence of A.
2) Let $Z = \{z_i\}_{i=1}^{n}$ denote the alphabet of the source sequence S of size N. Precisely, $S[i] \in Z$, for $i = 1, 2, \ldots, N$.
3) Let $T(\bullet)$ denote the encoding function for a prefix code. Precisely, given a symbol $b \in Z$, $T(b)$ returns the codeword of b. Further, let $\overline{T}(\bullet)$ denote the decoding function for a prefix code. Precisely, given a binary sequence B, $\overline{T}(B)$ is defined as:

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

where $z \in Z, 1 \leq r \leq |B|$.
4) The size of each block is denoted as t. In this paper, assume the blocks are cyclic. That is, if we have N blocks, then the i-th block refers to the $(i \pmod N)$-th block.
5) For a stack A, $A.pop(i)$ denotes the operation to pop the top i bits in the stack, and $A.push(X[i:j])$ denotes the operation to push $X[i:j]$ to the stack. $A.size()$ returns the size of the stack, and $A.popall()$ is the operation to pop all elements in the stack.

B. Canonical Huffman codes

A canonical Huffman code, introduced by Schwartz and Kallick [10], is a particular type of Huffman coding whose codebook can be encoded compactly. A property of canonical Huffman codes is that, a set of codewords with the same length is encoded to a set of successive binary integers [11]. This property allows us to determine the bit-lengths of a codeword by reading its prefix. For example, assume a canonical Huffman codebook is $a = 10, b = 0, c = 110, d = 111$. Then, we can determine the bit-length of a codeword by reading two bits at most.

C. Neighbor-based storage schemes

The following introduces the neighbor-based storage scheme [8]. The parameter t indicates the size of block.

1) If a sequence does not overflow, i.e., is shorter than t, it is placed in the corresponding block directly.
2) If a sequence overflows, i.e., is longer than t, its first t bits are stored in the current block, and its overflow part is stored in the next few neighboring blocks that have extra storage space.
3) Then overflows are stored based on the nearest-neighbor-first order.

Notably, in the proposed approach, the sequence refers to a codeword of the Huffman coding.

Algorithm 1 $E(t)$: Encoding algorithm

Input: A source sequence S of size N, and t
Output: X

```
Allocate Nt bits to X
Build a stack T_1
for i = 1 to N do
    C ← T(S[i])
    L ← |C|
    if L ≤ t then
        X[(i−1)t+1 : (i−1)t+L] ← C[1 : L]
        X[(i−1)t+L+1 : it] ← T_1.pop(t−L)
    else
        X[(i−1)t+1 : it] ← C[1 : t]
        T_1.push(C[t+1 : L])
    end if
end for
I ← 0
if T_1.size() ≠ 0 then
    repeat
        Store the T_1.popall() in the free space of i-th block
        I ← i
    until T_1.size() ≠ 0 or there are no free space
    Store the T_1.popall() in the tail of X
end if
return X and I
```

III. Encoding algorithm

In this paper, the proposed method adopts the codebook generated by canonical Huffman coding in Section II-B, and the bit ordering follows the neighbor-based storage scheme introduced in Section II-C.

To begin with, we create Nt bits space for X, and the code of $S[i]$ is stored in $X[it-t+1 : it]$, for $i = N, N-1, \ldots, 1$. If the code is shorter than t bits, it is stored in the corresponding block directly. However, if the code is longer than t bits, the first t bits of the code are stored in the current block, and the overflow part is stored in some free spaces in later blocks. The overflow parts of codewords can be stored in a stack until we find out free spaces.

Given the probability distribution of the sequence S, create a codebook T based on canonical Huffman coding. The stack T_1 is used to store the overflow bits. Algorithm 1 depicts the details. The integer I in algorithm 1 help us to find a proper block to start decoding, which means simplifies the process of decoding.

In order to facilitate the understanding, an example is given below to illustrate the procedure of encoding a sequence of symbols.

Example 1: We consider the alphabet $Z = \{a, b, c, d\}$ and the source sequence $S = (b, c, d, b, a)$. The codebook is $a \leftrightarrow 10$, $b \leftrightarrow 0$, $c \leftrightarrow 110$ and $d \leftrightarrow 111$. When $t = 2$, the symbol $S[1] = b$ is encoded to 0. As $t = 2$, 0 is stored in $X[1]$. Next, the symbol $S[2] = c$ is encoded to 110, so 11 is stored in $X[3:4]$ and put 0 into the stack. Next, $S[3] = d$ is encoded

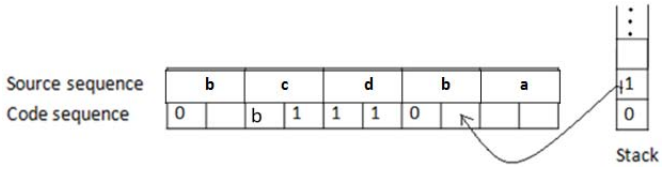

Fig. 1: Step 4 of Example 1

to 111, which is longer than t. Thus, 11 is stored in $X[5:6]$ and push 1 to the stack ($A.size() = 2$). Next, $S[4] = b$ is encoded to 0, so 0 is stored to $X[7]$. Then we have a free space, so $X[8]$ stores a pop bit from the stack (see Fig. 1). Next, $S[5] = a$ is encoded to 10 which are stored in $X[9:10]$. Finally, as $A.size() = 1$, the last bit of the stack is stored in the first free space $X[2]$, and output $X = 0011110110$ and $I = 1$.

IV. Symbol-retrieving algorithm

Given a code sequence X, the size of block t, the number of source symbols N and the index k, this section presents a method to retrieve the source symbol $S[k]$. Assume the last bit of the code $T(S[k])$ is stored in K-th block. The proposed algorithm in Section IV-A will decode a series of source symbols $S[k:K]$. However, if the codebook is based on canonical Huffman code, we can skip some bits of $X[(k-1)t+1:Kt]$. We proposed another retrieve algorithm in Section IV-B.

A. General condition

The retrieve algorithm is denoted by a function $(V, K, B) \leftarrow R_A(k)$, where $V = S[k]$, K is described above, and B denotes the unused code sequence in block k. Algorithm 2 depicts the details.

Algorithm 2 $R_A(k)$: Retrieve algorithm for prefix code

Input: A code sequence X of size M, t, N and k
Output: V, K, B

1: **if** $k > N$ **then**
2: **return** $(null, k, X[Nt+1:M])$
3: **end if**
4: $C \leftarrow X[(k-1)t+1:kt]$
5: **if** $\overline{T}(C) \neq NULL$ **then**
6: $(v, r) \leftarrow \overline{T}(C)$
7: **return** $(v, k, C[r+1, t])$
8: **else**
9: call $(V, K, B) \leftarrow R_A(k+1)$.
10: **end if**
11: $C \leftarrow (C|B)$
12: $k \leftarrow K$
13: Go to step 5

Example 2: Suppose we have a code sequence $X = (0,1,1,1,1,1,1,1,1,1,0,0,1,0,1,0)$, $N = 7$, and $t = 2$. The codebook is $a \leftrightarrow 10$, $b \leftrightarrow 0$, $c \leftrightarrow 110$ and $d \leftrightarrow 111$. If we want to retrieve $S[7]$, step 4 of $R_X(7)$ gives $C \leftarrow X[13:14]$, and thus $C = (10)$, which corresponds to a.

If we want to retrieve $S[5]$, step 4 of $R_X(5)$ gives $C \leftarrow X[9:10]$, and thus $C = (11)$. In this case, we call $(V_6, K_6, B_6) \leftarrow R_X(6)$, which returns $V_6 = B$, $K_6 = 6$ and $B_6 = (0)$. Then step 11 gives $C = (C|B) = (110)$, and thus we can decode the symbol c.

B. Using canonical Huffman code

By the property of canonical Huffman code, we can known its length by access its first few bits. So if the size of block is appropriate(big enough), then we can known the length of $T(S[k])$ by access the bits in k-block. In this case, the scheme to retrieve the k-th symbol is easier because we can skip some bits.

In algorithm 3, j donate $T(S[k])$ minus t and m used to count the bits we need to skip. When $m < 0$, the rest $|m|$ bits of current block is belong to $S[k]$. Algorithm 3 depicts the details.

Algorithm 3 $\overline{R}_A(k)$: Retrieve algorithm for canonical Huffman code

Input: A code sequence X of size M, N, t and k
Output: V

1: **if** $k > N$ **then**
2: **return** ERROR
3: **end if**
4: $C \leftarrow X[(k-1)t+1:kt]$
5: **if** $\overline{T}(C) \neq NULL$ **then**
6: $(v, r) \leftarrow \overline{T}(C)$
7: **return** v
8: **else**
9: $j \leftarrow T(S[k]) - t$
10: $m \leftarrow 0$
11: **if** $j + m > 0$ **then**
12: **while** $m \geq 0$ **do**
13: $k \leftarrow k + 1$
14: $m \leftarrow m + T(S[k]) - t$
15: **end while**
16: $C \leftarrow (C|X(kt + m - 1 : kt)$
17: **end if**
18: **end if**
19: $(v, r) \leftarrow \overline{T}(C)$
20: **return** v

If the prefix we need to read to longer than t, then we can not use the algorithm 3. But we also can skip some bits when we retrieve a symbol encoded by canonical Huffman code using algorithm 2. For instance, when we get the 6-th block $C = (00)$ in Example 1, we know the $T(S[k])$ because its first bit is 0. Thus, we can skip 1 bit.

V. Decoding algorithm

Given the M-bit code sequence X, the size of block t, and the number of source symbols N, the decoding procedure is presented in this section.

At first, we introduce an auxiliary scheme $F(s, e, T)$, where s and e are integers ($e \geq s$), T is a stack. The T store the overflowing bits which can not stored in the free space in s-th block to e-th block. The $F(s, e, T)$ decode the s-th symbol to e-th symbol in inverted order and return them. Algorithm 4 depicts the details.

Algorithm 4 $F(s, e, T)$: Decoding $S[s]$ to $S[e]$ in reversed order

Input: A code sequence X of size M, two integers $e \leq s$ and a stack T
Output: S

```
Built a char array S[e]
for i = e, e − 1, ..., s do
    C ← X[(i − 1)t + 1 : it]
    while T̄(C) = NULL do
        C ← (C|T.pop(1))
    end while
    (S[i], r) ← T̄(C)
    T.push(C[r + 1, t]
end for
return S
```

Given the auxiliary function $F(s, e, T)$. If $I = 0$, which means we do not use the circularity of blocks, we can easily decoding for end to head. Else, we need to use the auxiliary function twice. Algorithm 5 depicts the details. One can see this process visually in figure 2.

Algorithm 5 $D(I)$: Decoding algorithm

Input: A code sequence X of size M, I, N and t
Output: S

```
Built a char array S[N]
X_1 ← X[1 : Nt]
Build a stack T_1
if M > Nt then
    T_1.push(X[Nt + 1, M)
end if
if I = 0 then
    S ← F(1, N, T_1)
else
    Build a stack T_2
    S[1, I] ← F(1, I, T_2)
    while T_2 is note empty do
        T_1.push(T_2.pop(1))
    end while
    S[I + 1, N] ← F(I + 1, N, T_1)
end if
return S
```

Theorem V.1. *The algorithm 5 can correctly decode a code sequence X encode by scheme in Section V.*

Proof:. Because canonical Huffman code is prefix code, the code sequence C from k-th block can only be one of two cases:

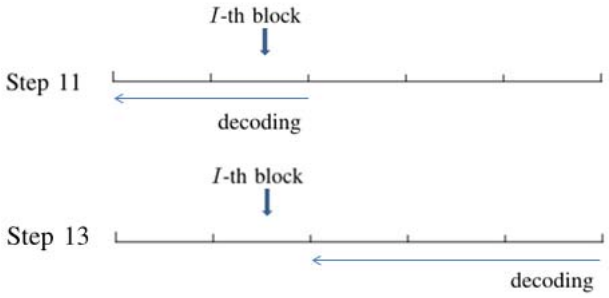

Fig. 2: Decomposition of the decoding process

1) $C[1, r-1]$ corresponds to $S[k]$, $C[r, t]$ consists of unused bits($C[r, L]$ can be NULL).
2) $(C|C_1)$ corresponds to $S[k]$, C_1 consists of unused bits in other blocks.

In case 1, it is obvious that the decoding scheme output the correct $S[k]$. In case 2, according to the encoding scheme, the C_1, e.q., the code sequence which C needed, was stored in the blocks after k-block. Thus, we do the decoding procedure from back to front and stored the unused bits in a stack. The property of stack(first in last out) make sure we always pop the right bit. □

Example 3: Suppose we have a code sequence $X = (0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0)$, $N = 7$, $I = 1$ and $t = 2$. The codebook is $a \leftrightarrow 10$, $b \leftrightarrow 0$, $c \leftrightarrow 110$ and $d \leftrightarrow 111$. To begin with, X is divided into two parts, $X_1 = (0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0)$ and $X_2 = (1, 0)$, so the stack $T_1 \leftarrow (01)$ and $T_2 \leftarrow$NULL. Next, we do $F(1, 1, T_2)$. The step 3 of algorithm 4 gives $C \leftarrow X_1[1 : 2]$ and thus $C = (01)$, 0 can be decoded to a unique symbol b. Thus, $S[1] \leftarrow b$, and the unused bit (1) are pushed onto the stack T_2. Then, the step 12 to step 14 give $T_1 \leftarrow (101)$.

Next, we do $F(2, 7, T_1)$. To decode the 7-th symbol, we let $C \leftarrow X_1[13 : 14]$ and thus $C = (10)$, which corresponds to the symbol a, and thus $S[7] \leftarrow a$. Next, to decode the 6-th symbol, we let $C \leftarrow X_1[11 : 12]$ and thus $C = (00)$, 0 can be decoded to a unique symbol b. Thus, $S[6] \leftarrow b$, and $T_1 \leftarrow (0101)$. Next, to decode the 5-th symbol, we let $C \leftarrow X_1[9 : 10]$ and thus $C = (11)$,which cannot be decoded to a unique symbol. Thus, a bit 0 is popped, and the new code is $C = (110)$, which gives $S[5] \leftarrow c$. One can follow the decoding algorithm to decode the whole sequence is $bdcdcba$.

VI. Analysis

Given the source sequence S of size N, then the cost of space is

$$\gamma_{\text{storage}} = \max\{\ell_{\text{traditional}}, Nt\}, \tag{2}$$

where $\ell_{\text{traditional}}$ is the length of canonical Huffman code of S. Thus, if $Nt \leq \ell_{\text{traditional}}$, the storage efficiency η_s is 1 because that canonical Huffman code is entropy code. Else,

we define η_s as the ratio between the entropy of the source and the total storage cost [8]. Then

$$\eta_s = \frac{H(z)}{t} = \frac{H(z)}{\frac{\ell_{\text{traditional}}}{N}} \cdot \frac{\ell_{\text{traditional}}}{Nt} = \frac{\ell_{\text{traditional}}}{Nt}, \quad (3)$$

where $H(z)$ is the entropy of a symbol z, $z \in Z$. So we should let $t \geq \lfloor \frac{\ell_{\text{traditional}}}{N} \rfloor$.

We define the cost of retrieving a symbol as the total number of bits to read. The cost of retrieving $S[i]$(assume the code of $S[i]$ is stored spanning over block i to block j) is

$$\gamma_{\text{retrieve}}(i) = l_i + \sum_{k=i+1}^{j} s_{\text{prefix}}(k), \quad (4)$$

where l_i is the length of the compressed codeword of $S[i]$, and $s_{\text{prefix}}(k)$ is the bits we need to read to know the length of $S[k]$. For instance, in example 1, $s_{\text{prefix}}(1) = s_{\text{prefix}}(4) = 1$ and $s_{\text{prefix}}(k) = 2$ for $k = 2, 3, 5$. Let $s_{\text{prefix}} = \max_{1 \leq k \leq N} \{s_{\text{prefix}}(k)\}$, then

$$\gamma_{\text{retrieve}}(i) \leq l_i + (w_i - 1)s_{\text{prefix}}, \quad (5)$$

where $w_i = j - i - 1$, e.q., w_i is the the number of blocks that $S[i]$ spans over. According to [8], we have (6) and (7): the expected cost of retrieving a symbol

$$\gamma_{\text{retrieve}} \leq \mathbb{E}(l) + (\mathbb{E}(w) - 1)s_{\text{prefix}}, \quad (6)$$

where $\mathbb{E}(w)$ is a non-increasing function of t.

Assume that the length of compressed symbol are i.i.d. distributed, we define $L_j = \sum_{i=1}^{j} l_i$, then

$$\mathbb{E}(w) \leq 1 + \sum_{j=1}^{N} P(L_j > jt). \quad (7)$$

Notice that we have a little difference in (7), this is because that scheme in [8] store 1 bit auxiliary data in the last bit of every block.

Example 1. *There is a particular example.*

Suppose we have a source sequence S, $S[k] \in Z = \{z_i\}_{i=1}^{n}$ and $P(z_i) = 2^{-i}$. Then the length of canonical Huffman code of z_i, denoted by $l(z_i)$, is i, for $1 \leq i \leq n-1$, and $l(z_n) = n - 1$. So the average length is $\frac{n+1}{2}$, i.e., $t \geq \lfloor \frac{n+1}{2} \rfloor$. Then

$$P(l(z_i) > t) \leq P(l(z_i) > \lfloor \frac{n+1}{2} \rfloor) \leq \sum_{i=\frac{n+1}{2}}^{n} 2^{-i} \leq 2^{-\frac{n+3}{2}} \quad (8)$$

Then $P(\mathbb{E}(w = 1)) \geq 1 - 2^{-\frac{n+3}{2}}$.

VII. Conclusions

We present a modified version of canonical Huffman coding such that the random access on the compressed file is allowed. Specifically, Example 1 shows that the probability of accessing a symbol directly is higher than $1 - 2^{-\frac{n+3}{2}}$. However, in conventional Huffman coding, it have to decompress a lot of bits to access a symbol. As the numbers of the occurrences of symbols are generally skew, we can improve random access even if the storage efficiency gets to 1. However, the analysis of storage efficiency in [8] has ignored the auxiliary bits. Actually, the proposed approach can be applied on any prefix variable-length encoding. Notice that the proposed method is to reorder the symbols in the code sequences without modifying the applied prefix encoding. This allows us to apply the proposed approach to any prefix encoding methods.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*. Prentice Hall Englewood Cliffs, 1990, vol. 348.

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

[3] E. HarpreetKaur, "A review of huffman coding of compression based methods in image segmentation," *International Journal for Technological Research In Engineering*, vol. 3, no. 9, 2016.

[4] M. Sharma, "Compression using huffman coding," *IJCSNS International Journal of Computer Science and Network Security*, vol. 10, no. 5, pp. 133–141, 2010.

[5] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, vol. 6, no. 2, pp. 163–180, 1985.

[6] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[7] G. Jacobson, "Random access in huffman-coded files," in *Data Compression Conference, 1992. DCC'92.* IEEE, 1992, pp. 368–377.

[8] 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.

[9] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

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

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