Bulletin of the University of Electro-Communications, 20-1·2, pp. 23-30 (2007) [Regular Paper]

A Post-Processing Mechanism for Sequential Use of Static and Dynamic Enumerative Code

Tsutomu Kawabata

Abstract

A bijection between a complete set of source words and a complete set of codewords defines a variable-to-variable length (VV) source code. Such code is used to parse sequentially a source sequence into codewords. In a naive parsing of a finite source sequence, the last incomplete source word requires a separate post-processing. However, if the sizes of the source and the code alphabet are the same and an end-offile is available, we show that there is an abstract and compact method for the post-processing. Furthermore, when a VV code is a concatenation of VF (variable-to-fixed length) source code and a complete FV (fixed-to-variable length) integer code, we propose a simple enumerative code implementation, which can be used for processing the last word before the end-of-file. This reduces the programming complexities compared with a naive post-processing. Furthermore, we apply the implementation to a dictionary trie based method for lossless data compressions, in particular, to the Ziv-Lempel incremental parsing algorithm. Finally, we extend the description in the binary alphabet to the one in a non-binary alphabet.

Keywords: variable-to-variable length source code, enumerative coding, Ziv-Lempel algorithm, post-processing issue.

1 Introduction

Consider source coding of source letter sequences to another letter sequences. In particular we consider the lossless coding, where encoding is one-to-one and hence there is a decoding rule which inverts the coded sequence into the original source sequence.

A practical design method of lossless source coding is to allow the decoder to keep track of the encoder's states. This method can be realized when the encoder employs relatively a small instantaneous code. The code parses an input source sequence into words and transforms each into a codeword. The decoder also employs the same instantaneous code which recovers the source word from the transmitted codeword. The codec, *i.e.*, the pair of the encoder and the decoder, can be dependent on the previously parsed words. Perfect recovery of the source word enables the codec to keep track of the same states. Thus both the encoder and the decoder can maintain, in particular, the same instantaneous code.

Each instantaneous code can be a variable-to-variable(VV) length code, and hence the above method allows a sequential and dynamic use of the VV code. A VV code is an one-to-one map between two complete codes. In such case, we can design the VV code in an elucidative principle of enumerative code[1] [2]. Namely, a VV coding can be designed in a concatenation of a variable-to-fixed length code and a fixed-to-variable length code. (For coding scheme terminology, we refer to [3])

However, one problem is that if the input length is arbitrary the last word may be incomplete and may need special post-processing. This will sacrifice coding rate and increase the complexity of the codec. In the following, we show that this problem can be resolved distinctly by an idea of enumerative coding. We will give an implementation which is commonly effective in

Received on August 6, 2007.
The University of Electro-Communications, Department of Information and Communication Engineerring

both the instantaneous code and the post-processing, thus both can be superposed. This will reduce the programming complexity due to the post-processing.

The organization of the paper is as follows. In Section 2, we introduce again the above motivation in a formal way, and show that the word left in an incomplete parsing can be encoded into an incomplete codeword, thus the decoder knows the occurrence of an end-of-file in the input. In Section 3, we show that an enumerative implementation of the function defined in Section 2 in the case of fixed, namely static, binary code. This section is an extraction of Section 4, thus can be skipped for reading. In Section 4, we show the enumerative implementation of the function defined in Section 2 in the case of the Ziv-Lempel incremental parsing. Here, the VV code is dynamic namely dependent on the previous parsing history. The section can be continued from Section 2 directly. In Section 5, we generalize the base of the algorithm from binary alphabet into non-binary alphabet. In the final section, we conclude.

2 A Scheme of VV Coding for File Transaction

In the following, denote by A a finite alphabet and fix during the following arguments. Let denote by A^l a set of length l words. In general, a superscript $*$ operated to an arbitrary set A, i.e. A^*, designates the set $\cup_{l=0}^{\infty} A^l$. Thus in our case, A^* consists of words of arbitrary length. Let $\sharp$ of a set denote its size. Hence, e.g., it holds that $\sharp A^l = (\sharp A)^l$. Thus the set A^0 consists only of an empty word λ.

A subset $T \subset A^*$ is called a trie, if any prefix of its element is also in T. For a trie T, we denote by ∂T the set $(\{\lambda\} \cup TA) - T$, which can be interpreted as the external leaves of T.

Let A be a binary alphabet $\{0, 1\}$. Then an example of T is $\{\lambda, 0\}$, and for this, $\partial T = \{00, 01, 1\}$.

Over the same alphabet, let T and U be tries of the same size. We assume that a bijection

$$c : T \cup \partial T \rightarrow U \cup \partial U$$

which satisfies

$$c(\partial T) = \partial U$$

can be constructed. Then necessarily a relation

$$c(T) = U$$

holds. Let c_0 denote the restriction of c to ∂T and let c_1 denote the restriction to T; that is $c_0 : \partial T \rightarrow \partial U$, and $c_1 : T \rightarrow U$. Since $A^* = (\partial T)^* T = (\partial U)^* U$, any $\mathbf{x} \in A^*$ is decomposed (or parsed) uniquely as $\mathbf{w}_1 \cdots \mathbf{w}_{k-1} \tilde{\mathbf{w}}_k$, where $\mathbf{w}_1 \in \partial T, \cdots, \mathbf{w}_{k-1} \in \partial T, \tilde{\mathbf{w}}_k \in T$. The output of the encoder

$$c : A^* \rightarrow A^*$$

for a finite input $\mathbf{x}$ is defined as

$$c(\mathbf{x}) = c_0(\mathbf{w}_1) \cdots c_0(\mathbf{w}_{k-1}) c_1(\tilde{\mathbf{w}}_k).$$

This output is uniquely decoded as follows provided an external file control, like the end-of-file is used. If the decoder completes parsing a codeword in ∂U, then it applies the function c_0^{-1} , and otherwise, it switches to apply c_1^{-1} . Here a point of the idea is that when we process an incomplete source word, we report the decoder by the combination of an incomplete codeword and the end-of-file.

The total code for the inputs of fixed length $n = |\mathbf{x}|$ is a concatenation of VV codes. For a fixed T, the overall rate of a FV (Fixed-to-Variable length) coding scheme is given by

$$\frac{(k-1)\log \sharp \partial T + \log \sharp T}{n \log \sharp A}.$$

A probabilistic analysis of this rate is related to the renewal theory, since the expected value of k is a renewal function (of n) for a memoryless source or a Markovian source.

3 Enumerative Implementation of the VV Code– A Binary Case–

The purpose of this section is to realize the idea of the previous section and construct a mapping c for a general VV code, in which the coding of c_0 and c_1 are unified and simplified.

We use the binary alphabet $A = \{0, 1\}$.

First we define c_0 so that $w \in \partial T$ and $c(w) \in \partial T$ have the same lexicographic order I in ∂T and ∂U respectively. Let the order start from 0 and hence I be in the range $0 \le I < N$, where $N = \sharp \partial T = \sharp \partial U$. Thus, c_0 can

be decomposed as $Dec\partial U \circ \mathrm{Enc}\partial T$, where

$$Enc\partial T : \partial T \to \{0, 1, \ldots, N-1\}$$

and

$$Dec\partial U : \{0, 1, \ldots, N-1\} \to \partial U.$$

$Enc\partial T(\mathbf{w})$ can be calculated by using an enumerative structure.

First we draw a trie $A^* = \{\lambda\} \cup (\{0\}A^*) \cup (\{1\}A^*)$ in our image by placing the root λ at the top, and all the other trie nodes down. A subtrie $\{0\}A^*$ represents the set of nodes which can be reached by descending from λ through the left branch with the label 0. Likewise a subtrie $\{1\}A^*$ represents the set of nodes which can be reached by descending from λ through the right branch with the label 1.

Let *left_leaves*(**v**) be a counter at **v**, which counts all the leaves, in ∂T, which have the common ancestor **v**0. Then the lexicographic order of w in ∂T is calculated, while descending T from λ along the path specified by w, as the sum of *left_leaves*(**v**) over every **v** for which **v**1 is on the path. Next, for a given $I = Enc\partial T(\mathbf{w})$, we can find $Dec\partial U(I) \in \partial U$, by descending from λ. Basic descending step at each node **v** is as follows. If $I \geq$ *left_leaves*(**v**), then we set new I to be $I -$ *left_leaves*(**v**) and takes the right branch. Otherwise we keep the same I and take the left branch. Finally, we arrive at the $Dec\partial U(I) = c_0(\mathbf{w})$, which becomes the VV codeword for w $\in \partial T$.

In a similar manner, we define c_0^{-1} as a composition $Dec\partial U \circ Enc\partial U$,
where

$$Enc\partial U : \partial U \to \{0, 1, \ldots, N-1\}$$

and

$$Dec\partial T : \{0, 1, \ldots, N-1\} \to \partial T.$$

Example 1: Let $T = \{0, \lambda, 10, 1\}$, where nodes are listed in the inorder, *i.e.*, in the order of leftleaves. Then $\partial T = \{00, 01, 100, 101, 11\}$, where the leaves are listed in the lexicographic order. Thus the counter values are *left_leaves*[0] = 1, *left_leaves*[λ] = 2, *left_leaves*[10] = 1, and *left_leaves*[1] = 2. Given a sequence 101, we set $I = 0$ and descend from λ down the trie along the sequence. We add *left_leaves*[λ] and *left_leaves*[10] to I and reaches to the leaf 101. Here we obtain $I = 3$, which is the lexicographic order of 101 in ∂T.

Let $U = \{00, 0, \lambda, 1\}$. Then $\partial U = \{000, 001, 01, 10, 11\}$. On U, the counter values are *left_leaves*[00] = 1, *left_leaves*[0] = 2, *left_leaves*[λ] = 3, and *left_leaves*[1] = 1. Given $I = 3$, we start from λ and descend the trie $U \cup \partial U$. Since I $\geq$ *left_leaves*[λ], we take the right branch to decode 1, and at this moment I becomes 0. Next we take the left branch and decode 0 and reaches to the leaf 10. Thus $c_0(101) = 10$. The calculation of c_0^{-1}goes similarly, only by changing the role of T and U. ■

Next we define c_1 so that each of v in T and $c_1(\mathbf{v}) \in U$ have the same inorders in T and U respectively. For T, the in-order $I(\mathbf{v})$ counts the leaves in ∂T which has the lexicographic order smaller than **v**. Thus the computation of c_1 follows the same procedure as c_0. Formally, the c is given as $DecU \circ EncT$, each of which is defined as follows.

$$EncT : T \to \{1, \ldots, N-1\},$$

and

$$DecU : \{1, \ldots, N-1\} \to U.$$

General procedure for $EncT$ is as follows. For a given $\mathbf{w} \in T$, we set $I = 0$ initially. Starting from the root λ we descend T down to **w**. At a node **v** we add *left_leaves*(**v**) to I when we take the right branch. When we arrived at **w**, we add *left_leaves*(**w**) to I and exit. This is equivalent to the situation of taking the right branch and after that taking all left branches until it reaches to a leaf in ∂T. After we exit, we move to the U and start the procedure $DecU$. For a given I, we find $\mathbf{u} \in U$ whose in-order is I, as in the following. We decode each bit of **u** while descending from λ to **u**. At each descending step at a node **v**, we attempt to subtract *left_leaves*(**v**) from I, as long as the result is non-negative. If the result is positive, we have a new value for I, and takes the right branch. If it is negative, we keep the value of I unchanged and take the left branch. If the resulted I value is zero, then we stop and output **v** as $c(\mathbf{w}) \in U$.

Example 2 : Let the input be 10 followed by the end-of-file. Setting $I = 0$, we start to descend T from λ. We add *left_leaves*[λ] = 2 to I and reach the node 10. Here we read the end-of-file. The algorithm terminates after adding *left_leaves*[10] = 1 to I. Thus I becomes 3, which is the in-order of 10 in T. Next in U, we start

with I = 3 and attempt to descend U from λ. Since I = *left_leaves*[λ], subtraction makes I zero, thus we stop and exit, with the output $c_1(10) = \lambda$. In the decoder, we input λ followed by the end-of-file. Thus we switch to $c_1^{-1}()$. However,we do not need to know the existence of the end-of-file. Actual steps are as follows. On U, we set $I = 0$ and starts descending from λ. Since we read the end-of-file, we exit the trie U, after adding *left_leaves*[λ] = 3 to I. Next on T, in the decoder's second step we starts from $I = 3$ and descend T from λ while decoding each bit. In the decoder, we first compare I with *left_leaves*[λ] = 2, and output the bit 1, and set I = 1. Then we decode the bit 0, and leave the value I unchanged. Next we attempt to output the bit 1. However at this point I becomes zero. Thus we simply stops. The output thus becomes 10. ■

4 Application to Ziv-Lempel Incremental Parsing

The scheme used so far can be extended to a dynamical case, that is, to the case that the set T and U are dependent on the past history. A good example is the Ziv-Lempel incremental parsing. In this section, we concede the subject to the nature in the algorithm description. Thus the name of a procedure/circuit will be the subject of the operations in its body.

The Ziv-Lempel incremental parsing algorithm [4] decomposes a sequence of letters from left to right into words such that a new word is the shortest one which differs from any of the previously parsed words. It organizes the parsed words into a trie (which we call a dictionary trie), and based on which it encodes/decodes the new word[6][5]. (Due to this fundamental simplicity, the coding rate and the redundancy is analyzed deeply not only in Shannon theoretical studies, *e.g.* [2][9][13], but also in algorithm analyses, *e.g.* [11][14][12].) This trie has the set of external leaves, that corresponds to the set of possible new parsing words. An algorithm proposed by [8] calculates an enumerative index[2] of the new word as the lexicographic order of the word in the set of external leaves.

To argue more in detail, let us consider a binary source alphabet and a binary code alphabet. Thus the enumerative index must be encoded into a binary sequence. The encoder does parsing in two phases, as in IntEnc ◦ DicEnc,
where

$$\text{DicEnc} : \{0, 1\}^* \rightarrow \mathcal{N} \cup \{0\} \qquad (1)$$

$$\text{IntEnc} : \mathcal{N} \cup \{0\} \rightarrow \{0, 1\}^*, \qquad (2)$$

where we denoted by $\{0, 1\}^*$ the set of all binary sequences and by $\mathcal{N}$ the set of natural numbers.

Correspondingly, decoder recovers the parsing as in DicDec ◦ IntDec, where

$$\text{IntDec} : \{0, 1\}^* \rightarrow \mathcal{N} \cup \{0\} \qquad (3)$$

$$\text{DicDec} : \mathcal{N} \cup \{0\} \rightarrow \{0, 1\}^*. \qquad (4)$$

Here DicEnc/DicDec abbreviates a pair of dictionary source codecs, and IntEnc/IntDec does a pair of integer source codecs.

Example 3: Let us give an example of the above scheme. Let an input be 001. The algorithm sets {$\mathbf{w}_0 = \lambda$} as an initial dictionary trie. The associated external leaves are {0, 1}.

Now DicEnc first reads 0, and finds a new word $\mathbf{w}_1$ = 0. DicEnc outputs $I = 0$ as an enumerative index of $\mathbf{w}_1$ in {0, 1}. The index takes values in the range $0 \le I < \#\{0, 1\}$. The dictionary trie becomes the set {$\mathbf{w}_0 = \lambda$,$\mathbf{w}_1$ = 0}, with the set of external leaves {00, 01, 1}. IntEnc reads I and outputs its integer codeword, *i.e. a binary representation*. At the decoder, IntDec reads the integer codeword and returns I, and from which DicDec reproduces 0.

In the next parsing, DicEnc reads 01, and finds $\mathbf{w}_2$ = 01, and outputs $I = 1$ as an enumerative index of $\mathbf{w}_1$ in {00, 01, 1}. The index is in $0 \le I < \#\{00, 01, 1\}$. The IncEnc takes I as its input and outputs its integer codeword. The dictionary trie becomes {w0 = λ ,$\mathbf{w}_1$ = 0,$\mathbf{w}_2$ = 01} with the set of external leaves {00,010,011,1}. At the decoder, DicEnc takes the codeword as its inputs and returns I, and from which DicDec reproduces 01. ■

In the work [8], a detailed implementation of DicEnc/DicDec is given. The main idea of the implementation is to introduce an enumerative mechanism in the dictionary trie. At every inner node, a counter keeps the number of external leaves of its left subtrie. (The counter has been named as *left_leaves*.) DicEnc calculates the variable I while descending the trie along a new word $\mathbf{w}$, such that I is the sum of the counters associated with the inner nodes for which the next bit is 1. DicEnc maintains the counter values at the same time. DicDec receives I and outputs $\mathbf{w}$. The idea of the procedure is as follows.

Starting from the root while descending along the unknown w, at each inner node **v**, DicDec holds the value I as the enumerative index of the remaining suffix of **w** in the external leaf set of the subtrie with the root **v**. At the inner node **v** DicDec attempts to subtract left_leaves from I. The next bit must be 1 if the subtraction is possible and 0 otherwise, in the latter case I is left unchanged. For example I is annihilated at the moment when DicDec decodes the last 1. DicDec easily maintains the counters at the same time.

However, the work [8] describes DicEnc/DicDec but not IntEnc/IntDec, and does not mention the case when the input does not finish on the parsing boundary.

Let us explain this below. At each parsing, IntEnc/IntDec knows that the index I is in a range, say, $0 \le I < \rho$. In such a case, a typical IntEnc/IntDec employs the set of ρ leaves of a complete binary tree with the maximal depth $k = \lceil \log_2 \rho \rceil$ and with the minimal depth at least $k - 1$ [7]. However, if the last word ends off at an inner node, then the IntEnc has no way to tell to IntDec about this event due to the compactness of the code.

The remedy to this problem is the main purpose of this study. If we assume a complete binary tree code for IntEnc/IntDec, it is imagined that there is no elegant solution, if the total code is restricted to an instantaneous code. (A naive post-processing method, which does completing the last word and making correction, would require an extra routine and additional information bits.) However, as far as the codecs are allowed to use the end-of-file, which is available when the file transmission is managed in a different level, a solution is found. (This is the case in usual applications.) Let the dictionary trie consist of a set of inner node T, let the IntEnc/IntDec tree consist of a set of inner node U, and let assume that both of T and U are the same size. Then there is a bijection $c_1 : T \to U$, as well as a bijection between the two sets of corresponding leaves. Thus an image $c_1(\mathbf{v})$ of an incomplete source word **v**, transmitted with an end-of-file, can report the decoder on this incomplete event, and can activate the inverse $c_1^{-1} : U \to T$.

Now, based on the above idea, we go more in detail. Since DicEnc/DicDec already have prepared a counter at the inner node in the dictionary trie[8], the in-order becomes a good candidate for making the bijection. The in-order of an inner node is defined as the lexicographic order of the leftmost external leaf, in the set of external leaves, of the right subtrie of the inner node. This is just the value of I plus the counter at the inner node. Equivalently it is the value of I that DicEnc will calculate when it further reads 1. Thus, on detecting the end-of-file, the IntEnc switches and inputs the sum of I and the counter. IntEnc outputs the associated complete binary tree codeword and deletes from which the suffix 10^* (meaning 1 followed by arbitrary number of zero). The encoder sends this output.

Before we go to the decoder, we pause to give some remarks. First, IntEnc/IntDec can employ the same type of trie structure as DicEnc/DicDec does. Next, we note that a suffix can be deleted always, *i.e.*, 1 always exists in the complete binary tree codeword, since the integer for IntEnc can never take zero. This is due to the following fact for binary complete trees that the number of inner nodes are one less than the number of leaves.

Now, the decoder does the converse. First IntDec will fail to find a leaf of a complete codeword. IntDec appends a suffix 10^* to a received bits to obtain a complete codeword, which represents the in-order of the inner node. IntDec informs DicDec of the value I and switch DicDec into the function c_1^{-1} . However the difference in the procedure is minimal. In fact, DicDec starts to decode I as the normal case, and simply stops just before decoding the last 1, *that is*, the moment when I is annihilated.

Example 4: We give a simple example. Let 01 and an end-of-file follows the inputs in the previous example. On encountering the end-of-file while decoding the t = 3rd word, DicEnc will output (I = 1) + ($left\_leaves$ = 1) for IntEnc and report on the occurrence of the end-of-file. The complete binary tree for $0 \le I <$ (3+1) has the external leaves {00, 01, 10, 11}, among which IncEnc will select 10 as usual, but will delete a suffix 10. Thus, IntEnc will output λ and transmit with the end-of-file. Based on this the decoding starts. The IntDec encounters the end-of-file after λ. Since λ apparently is not a complete code word, the IntDec appends 10 to the λ to recover an integer I = 2 and reports DicDec that it should be interpreted as an in-order. DicDec then starts decoding from I = 2 as usual, outputs 01, and attempts to decode another 1. However, since I would then be

annihilated, DicDec stops there. ■

The detailed implementation is given in Appendix[16].

5 Application to the Non-Binary Alphabet Sources

The basic scheme given in Section 2 holds for any finite alphabet. The enumerative algorithm designed for the binary case almost apply to the finitealphabet case. In this section we study the enumerative algorithm for them-ary case and give remarks on the difference from the binary case.

Let the source alphabet be $A = \{0, 1, \ldots, m-1\}$, thus T is an m-ary tree. Let the code alphabet be $B = A$. Thus U is also an m-ary tree. The enumerative implementation of Enc∂T/Dec∂T is mostly the same. The only difference is that we prepare a counter on every $\mathbf{v} \in T$ which holds the number $L[\mathbf{v}]$ of external leaves, for which $\mathbf{v}$ is the common ancestor. The basic computation is as follows. We start with $I = 0$ and from the root node λ . While descending along the word $\mathbf{w}$ we are parsing, at every inner node $\mathbf{u} \in$ T for which $\mathbf{u}k$ is a prefix of $\mathbf{w}$, we add to I the cumulative counts

$$C_k := \sum_{l=0}^{k-1} L[\mathbf{u}l].$$

Thus the basic procedure for decoding k at $\mathbf{u}$ is to locate I, to a level in a staircase whose step sizes given by a list $\{L[\mathbf{u}0], L[\mathbf{u}1], \ldots, L[\mathbf{u}(m-1)]\}$.

Next, we consider the post-processing case. A bijection $T \to U$ can be obtained similarly. The in-order of $\mathbf{v}$ in T is equated with the in-order of the corresponding $\mathbf{u}$ in U. Thus let us see how the in-order can be calculated. The in-order J of $\mathbf{v} \in T$ is calculated as follows. Starting from the root we first initialize J to zero. While descending a parsed word, at every inner node $\mathbf{v}$, we acquire the in-order $(J + 1) + (L[\mathbf{v}0] - 1)/(m - 1)$, where the second term means the number of inner nodes of the subtrie with the root $\mathbf{v}0$, and 1 in the first term is for the node itself. Thus when we descends the path $\mathbf{v}k$ we should update J by adding a number $(C_k - k)/(m - 1)$, plus 1 only when $k \geq 1$. In the binary case, the calculation is simplified, as was described before.

In the above, we assumed implicitly that m is small. When m is large, we should reduce the complexity for computing C_k. In the case of $m = 2^d$, we can introduce a binary complete trie of depth d. Let $b_1^d(k) \in \{0, 1\}^d$ denote a binary representation of k. For each $\mathbf{c} \in \cup_{l=0}^{d-1} \{0, 1\}^l$, we associate an integer $C[k]$ holding the sum of all the L_k for which $b_1^d(k)$ has a prefix $\mathbf{c}0$. Then C_k can be calculated, while changing $\mathbf{c}$ over the proper prefix of $b_1^d(k)$, by accumulating $C[\mathbf{c}0]$ only when $\mathbf{c}1$ is a prefix of $b_1^d(k)$.

6 Conclusion

Enumerative method is useful in lossless and lossy data compressions. We illustrated this method in the implementation of sequential use of lossless Variable-to-Variable(VV) code. One of a practical problem in the VV code is the post-processing problem. We showed that the method of enumerative coding is still effective in the detailed design of VV code, by showing that transactions of complete code and that of incomplete code can be superposed and hence that programming complexity is apparently reduced, provided that an end-of-file character is available. We have first shown this method in the static VV code and then further applied to a dynamic VV code, in particular to the Ziv-Lempel incremental parsing algorithm. Although all the above were illustrated in the binary alphabet source, we noticed on the extension to the non-binary alphabet.

Acknowledgments

The author thanks the anonymous reviewer for careful reading and the comments which improved the manuscript.

References

[1] J. P. M. Schalkwijk: An Algorithm for Source Coding, IEEE Trans. on Information Theory, 18, 395-399(1972).

[2] Cover, T. M. : Enumerative Source Coding, IEEE Trans. on Information Theory, 19,1,73-77(1973).

[3] Pasco, R.: Source coding algorithms for fast data compression, Ph.D Dissertation, Stanford University (1976).

[4] Ziv, J. and Lempel, A.: Compression of Individual Sequences via Variable-rate Coding, IEEE Trans. on Information Theory, IT-24, 5, 530-536 (1978).

[5] Ma, J. S.: Data Compression, Ph.D Dissertation, Dept. Electrical and Computer Eng., Univ. of Massachusetts, Amherst (1978).
[6] Rissanen, J.: A Universal Data Compression Systems, IEEE Trans. on Information Theory, IT-29, 5, 656-664, (1983).
[7] Bell, T. C., Cleary, J. G., and Witten, I. H.: *Text Compression*, Prentice Hall(1990).
[8] Kawabata, T. and Yamamoto, H.: A New Implementation of Ziv-Lempel Incremental Parsing Algorithms, IEEE Trans. on Information Theory, 37,5, 1439-1440,(1991).
[9] Plotnik, E., Weinberger, M., and Ziv, J.: Upper bounds on the probability of sequences emitted by finite state sources and on the redundancy of the Lempel-Ziv algorithm, IEEE Trans. on Information Theory, 38, 2, 66-72, (1992).
[10] Sheinwald, D.: On Binary Alphabetical Codes, Proceedings of the Data Compression Conference, IEEE Computer Society Press, 112-121,(1992).
[11] Kawabata, T.: Exact Analysis of the Lempel-Ziv algorithm for I.I.D. source, IEEE Trans. on Information Theory, 39, 2, 698-702, (1993).
[12] Jacquet, P. and Szpankowski, W.: Asymptotic behavior of the Lempel- Ziv parsing scheme and digital search trees, Theoretical Computer Science, 144, 161-197 (1995).
[13] Savari, S. A.: Redundancy of the Lempel-Ziv incremental parsing rule, IEEE Trans. on Information Theory, 43, 2, 9-21, (1997).
[14] Kawabata, T.: A Note on a Sequence Related to the Lempel-Ziv Parsing, IEICE Trans. on Fundamentals, E83, 10, 1979-1982, (2000).
[15] Cover, T. M. and Thomas, J. A.: *The Elements of Information Theory*, 2nd. Ed., Wiley(2006).
[16] Kawabata, T.: A Post-Processing for the Enumerative Code Implementation of Ziv-Lempel Incremental Parsing, IEICE Trans. on Communications, E90,11,3263-3265,(2007).

Appendix to §4[16]

In the following, we describe a part of the algorithm. There are four procedures **DicEnc**,**IntEnc**,**IntDec**, and **DicDec**. Those were defined in the main section.

Algorithmic descriptions of **DicEnc** and **DicDec** appeared in [8] first. In this appendix we modify those according to our current purpose. **DicEnc** depends on **IntEnc**, and **DicDec** depends on **IntDec**. The functions of **IntEnc** and **IntDec** are apparent, thus the detailed descriptions are omitted.

Let us describe the data structure, of the dictionary trie. Each node has a record with three fields, called *left_leaves*, *left* and *right*, which designate the number of leaves of the left subtree, the pointer to the left child, and to the right child, respectively. The procedure *make* generates the node, by initializing the *left_leaves* to unity and the other two pointers both to **nil**s, and finally returns the pointer to the node. Both *root* and *q* are pointers to the node, and *r* is an indirect pointer(*i.e.* the pointer to the pointer) to the node.

We denote the destination of a pointer by the pointer name followed by ↑. A complete binary tree is represented by the set of inner nodes. Therefore, in our algorithm no leaves are actually created. The encoder will first initialize the root node and then repeat the parsing process. For each repetition, q will descend a path from the root down to the leaf, as directed by the value of the input bit. We use r to refer indirectly to the left or the right child of the node that q points to, depending on whether the input bit is zero or one. The variable I, initialized to zero, is used to calculate the enumerative index of w by summing up the *left_leaves* of the node, only when the input bit is 1. When the input bit is 0, we just add one to the *left_leaves* of the node. When q becomes **nil**, this means that we have reached a leaf, thus that a parsing has completed.

Let $CBT : \{I : 0 \le I < \rho\} \rightarrow \{0, 1\}^*$ represent a procedure, for a known ρ, which returns a complete binary tree representation of I. The method is described in Appendix A.2. of [7]. Let an inverse of CBT be given by $CBT^{-1} : CBT(\{I : 0 \le I < \rho\}) \rightarrow \mathcal{N} \cup \{0\}$.

In the following, IntEnc(I,Final) simply returns $CBT(I) \in \{0, 1\}^*$ but with deleting a postfix 10^* only when the logical variable Final is true. For CBT normal codewords, IntDec(**var** Final) sets Final ← **false** and applies

$CBT^{-1}()$. For a CBT incomplete node, it sets Final ← **true**, and then completes the input sequence into a CBT codeword by appending a postfix 10^*, and applies $CBT^{-1}()$ to the modified inputs.

DicEnc:

```
make(root);
repeat
    q ← root; I ← 0;
    repeat
        if get(bit) =EOF then
            IntEnc(I+q↑.left_leaves,true);
        else if bit = 0 then
            begin
                q↑.left_leaves ← q↑.left_leaves + 1;
                r ← the pointer to q↑.left;
            end
        else
            begin
                I ← I + q↑.left_leaves;
                r ← the pointer to q↑.right;
            end;
        q ← r↑ ;
    until q=nil;
    IntEnc(I,false);
    make(r↑ );
until false;
```

DicDec:

```
make(root);
repeat
    q ← root; I ← IntDec(Final);
    repeat
        if q↑.left_leaves > I then
            begin
                output(0);
                q↑.left_leaves ← q↑.left_leaves + 1;
                r ← the pointer to q↑.left;
            end
        else if(q↑.left_leaves <I)or not Final then
                begin
                    output(1);
                    I ← I – q↑.left_leaves;
                    r ← the pointer to q↑.right;
                end;
        else Exit();
        q ← r↑ ;
    until q=nil;
    make(r↑ );
until false;
```