HOKKAIDO UNIVERSITY

Title	An Efficient Algorithm for Almost Instantaneous VF Code Using Multiplexed Parse Tree
Author(s)	Yoshida, Satoshi; Kida, Takuya
Citation	2010 Data Compression Conference (DCC), 219-228 https://doi.org/10.1109/DCC.2010.27
Issue Date	2010-03-24
Doc URL	http://hdl.handle.net/2115/46943
Rights	© 2010 IEEE. Reprinted, with permission, from Yoshida, S., Kida, T., An Efficient Algorithm for Almost Instantaneous VF Code Using Multiplexed Parse Tree, 2010 Data Compression Conference (DCC), Mar. 2010. This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Hokkaido University products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
Type	proceedings (author version)
File Information	DCC2010_219-228.pdf

Instructions for use

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

An Efficient Algorithm for Almost Instantaneous VF Code Using Multiplexed Parse Tree

Satoshi Yoshida† Takuya Kida†
†Graduate School of Information Science and Technology, Hokkaido University.
Kita 14-jo, Nishi 9-chome, Kita-ku, 060-0814 Sapporo, Japan
{syoshid,kida}@ist.hokudai.ac.jp

Abstract

Almost Instantaneous VF code proposed by Yamamoto and Yokoo in 2001, which is one of the variable-length-to-fixed-length codes, uses a set of parse trees and achieves a good compression ratio. However, it needs much time and space for both encoding and decoding than an ordinary VF code does. In this paper, we proved that we can multiplex the set of parse trees into a compact single tree and simulate the original encoding and decoding procedures. Our technique reduces the total number of nodes into $O(2^\ell k - k^2)$, while it is originally $O(2^\ell k)$, where ℓ and k are the codeword length and the alphabet size, respectively. The experimental results showed that we could encode and decode over three times faster for natural language texts by using this technique.

1 Introduction

From the viewpoint of speeding up compressed pattern matching, *variable-length-to-fixed-length codes* (VF codes for short) are reevaluated recently [KS09, Kid09]. A VF code is a coding scheme that parses an input text into a consecutive sequence of substrings by using a dictionary tree, which is called a parse tree, and then assigns a fixed length codeword to each substring. *Tunstall code* [Tun67], which is a classical and typical VF code, is proved to be an entropy code for a memory-less information source (see also [Sav98]); its average of code length par symbol comes asymptotically close to the entropy of the source when the code length goes to infinity. However, its actual compression ratio is rather worse in comparison with Huffman code.

Several improvements on VF codes have been proposed so far. *Almost Instantaneous VF code* (AIVF code for short[1]) proposed by Yamamoto and Yokoo [YY01] is one of the most attractive codes from the practical views. Its compression ratio is almost comparable to that of Huffman code. In fact, even if the codeword length is short, AIVF code achieves a better compression ratio than Tunstall code whose codeword length is considerably long (see Section 4). However, its encoding and its decoding speeds are slower. The reason is that we must construct $k-1$ parse trees when the alphabet size of the text is k, which usually reaches to $70 \sim 140$ for natural

[1] "Almost instantaneous (VF) code" means originally that we need one symbol read-ahead for encoding. However, we use it as the unique name in this paper.

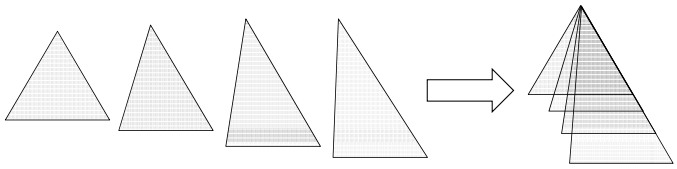

Figure 1: Multiplexing the parse trees of AIVF code into a single tree

language texts. Of course, we also need much memory space to hold the trees. These weaknesses prevent us from doing fast pattern matching on AIVF code. Therefore, VF code with a more compact parse tree that can also achieve a good compression ratio comparative to AIVF code is strongly desired.

In this paper, we propose an efficient algorithm for AIVF code, which integrates the multiple parse trees into a compact single tree and simulates the encoding and the decoding procedure of the original AIVF code. Our idea comes from the observation that many nodes in multiple parse trees of an AIVF code are common and thus they can be multiplexed (see Fig. 1). We named the integrated parse tree as *Virtual Multiple AIVF parse tree* (VMA tree for short). We present that the size of a VMA tree is $O(2^\ell k - k^2)$ while that of the original is $O(2^\ell k)$. In other words, the total number of nodes which must be constructed can be reduced by $\Omega(k^2)$. We also present that our technique enables us to encode and decode over three times faster for natural language texts in actual.

2 Preliminaries

2.1 Terminology and Notation

Let Σ be a finite alphabet and Σ^* be the set of all strings over Σ. We denote the length of string $x \in \Sigma^*$ by $|x|$. We call the string whose length is 0 the *empty string* and we denote it by ε. Therefore, $|\varepsilon| = 0$. The concatenation of two strings x_1 and $x_2 \in \Sigma^*$ is denoted by $x_1 \cdot x_2$, and also write it simply as $x_1 x_2$ if no confusion occurs.

We denote the occurrence probability of string $x \in \Sigma^*$ in a text S by $\Pr_S(x)$. We define $\Pr_S(\varepsilon) = 1$ for convenience. Although $\Pr_S(x)$ depends on S, we write it simply as $\Pr(x)$ when the target text is obvious from the context or when we deal it as the statistical feature of a given information source.

We call a tree in which each node has at most k children, a *k-ary tree*. We call a node which has children an *internal node* (or inner node), and call a node which has no children a *leaf node* (or leaf). We also call the node which has no parent (i.e. the top of a tree) the *root node* (or the root). Furthermore, in a k-ary tree, we call a node which has exactly k children a *complete internal node*, and also call an internal node which is not complete an *incomplete internal node*. We call a tree in which all internal nodes are complete, a *complete k-ary tree*.

For a tree (or a forest) T, We denote the set of all leaves in T by $\mathcal{L}(T)$, and denote the set of all incomplete internal nodes in T by $\mathcal{I}(T)$. We also denote the union of

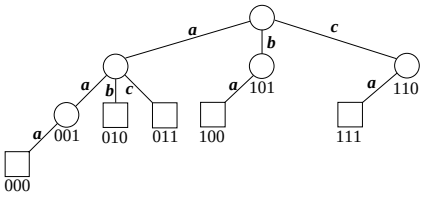

Figure 2: Parse tree for AIVF code

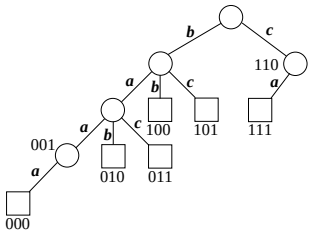

Figure 3: Another parse tree

$\mathcal{L}(T)$ and $\mathcal{I}(T)$ by $\mathcal{N}(T)$, and denote the size of set S by $\#S$. Then, for example, we can denote the number of leaves by $\#\mathcal{L}(T)$. For a node n, we call the number of children of n the *degree* of n, which is denoted by $d(n)$.

2.2 Almost Instantanenous VF code

In this section, we give a brief survey of AIVF code [YY01]. AIVF code is based on Tunstall code [Tun67]. In order to improve its compression ratio, Yamamoto and Yokoo employed two techniques: one is to assign codewords onto the incomplete internal nodes in the parse tree, and the other is to parse a text by using multiple trees.

Let $\Sigma = \{a_1, \ldots, a_k\}$, and assume that all symbols in Σ are sorted by their occurrence probabilities for convenience of discussion. That is, $i < j$ implies $\Pr(a_i) \geq \Pr(a_j)$. We also assume that all codes discussed below are binary codes.

2.2.1 Improvement by assigning codewords to internal nodes

For a given text whose alphabet size is k, Tunstall code uses a complete k-ary tree as a parse tree, which is called a Tunstall tree. Each edge in the tree is labeled with a symbol in Σ. Each node corresponds to a string over Σ which is spelled out from the root to the node. Each codeword, which is a binary string of length ℓ, is assigned to each leaf in the Tunstall tree, namely, there is a one-to-one correspondence between a codeword and a leaf. A given text is parsed into a consecutive sequence of blocks by the tree, and each block is encoded with the corresponding binary codeword.

Note that unused codewords of length ℓ exist if $\#\mathcal{L}(T) \neq 2^\ell$. This suggests that we can make the average block length longer by assigning such unused codewords to some strings. If we add a leaf to a complete k-ary tree, an incomplete internal node is made. That is, this also suggests that we can make the average block length much longer if we can remove low-frequency leaves and extend useful edges.

Figure 2 is an example of the parse tree for a AIVF code, where $\Sigma = \{a, b, c\}$ (i.e. $k = 3$) and the codeword length ℓ is equal to 3. We will discuss later how to build the optimal parse tree in the sense of maximizing the average block length. If we have the parse tree, the encoding procedure is as follows: Reading a symbol one by one from the text, we traverse the tree from the root; if we can not traverse any more, we output the codeword assigned at the current node, and then return to the root and continue traversing.

Now we consider to encode a given text $S = aabaabaccab$ by using the parse tree in Fig. 2. Then, S is split into blocks $aa \cdot ba \cdot ab \cdot ac \cdot ca \cdot b$ by the tree, and thus, the output binary sequence becomes $001 \cdot 100 \cdot 010 \cdot 011 \cdot 111 \cdot 101$. Its length is 18 bits.

Next we will discuss about the construction of the optimal parse tree. Let T be the parse tree for AIVF code. Let C_T be the set of all strings which are assigned codewords in T. Note that C_T is regarded as a dictionary which consists of strings registered to T. Let M be the number of codewords. Then, our aim is to construct an optimal parse tree that maximizes the average block length $\sum_{x \in C_T} |x| \cdot \Pr(x)$ for a memory-less information source.

Although we omit the detail, a greedy algorithm brings about the optimal solution in this case. It is proved that exchanging a node in the parse tree constructed by the algorithm below for another node does not increase the average block length[YY01]. The strategy of the algorithm for constructing the optimal parse tree is to continue to add the best node one by one that maximizes the average at each stage. Note that, however, when the $k-1$th child of an incomplete node is going to be added, we can increase the average block length by adding also the kth child and assigning a codeword onto it (i.e., by making the internal node complete). The reason is that we do not assign a codeword to a complete internal node.

Therefore, the outline of the optimal parse tree construction algorithm is as follows. Recall that we assume $\Pr(a_i) \geq \Pr(a_j)$ for $i < j$. We identify a node with its corresponding block below.

1. Create an initial tree T^* which consists of the root node and its k children, and let $m = k$.
2. Find the node which has the maximum probability, namely, $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n)$.
3. Let S_1 be the growth of the average block length after making $\hat{n}$ complete.
4. Let $S_2 = 0$. Repeat the following $k - d(\hat{n}) - 1$ times.
   (a) Find the best position where the next node must be added, namely, $\tilde{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$.
   (b) Add the growth of average block length after creating the $d(\tilde{n}) + 1$th child of $\tilde{n}$ to S_2.
5. If $S_1 \geq S_2$, make $\hat{n}$ complete. Otherwise, repeat following $k - d(\hat{n}) - 1$ times. $\tilde{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$, and add the $d(\tilde{n}) + 1$th child of $\tilde{n}$.
6. Add $k - d(\hat{n}) - 1$ to m. Repeat the above four steps until $k - d(\hat{n}) - 1 \leq M - m$
7. Repeat following $M - m$ times. $\tilde{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$, and add the $d(\tilde{n})+1$th child of $\tilde{n}$.

For the decompression, we need the same parse tree used in the encoding. All we need for reconstructing it is only the information about the frequencies of symbols. After we reconstruct the parse tree, we can decode by mapping each codeword to a string registered in the tree. We omit its detail for lack of space.

2.2.2 Improvement by using multiple parse trees

In AIVF code, the blocks are not statistically independent even if the information source is memory-less. Parsing with a parse tree discussed above causes contexts between blocks. Yamamoto and Yokoo also present that we can make the average block length longer by using a set of parse trees in order to catch the contexts.

For example, assume that block aa is currently parsed and 001 is output while we encode with a parse tree shown in Fig. 2. In this case, the next symbol is b or c, because we had to continue traversing if the next symbol was a, and thus, block aaa should be parsed and 000 should be output. Therefore, when 001 is output, the node corresponding the code $000, 001, 010, 011$ are unreachable. This suggests that we can make the average block length longer by assigning such unreachable codewords to other strings. For the running example, instead of using the tree in Fig. 2, we can make the next block longer by using the tree in Fig. 3 in this case.

In the method proposed in [YY01], $k-1$ parse trees T_i $(i = 0, \ldots, k-2)$ are utilized. For each i, the ith parse tree T_i has the root and its children labeled by $a_{i+1}, \ldots, a_k$ $(i = 0, \ldots, k-2)$. (Recall that $\Pr(a_i) \geq \Pr(a_j)$ for $i < j$.) That is, we encode and decode a given text switching the parse trees according to the context.

For each T_i, we construct the optimal parse tree in almost the same way as we mentioned above. Only the numbers of children under the roots are different each other. Note that $\#\mathcal{N}(T_i) = M$ for each T_i, where M is the number of codewords. Therefore, the tree T_i becomes sharper and taller for larger i. These multiple parse trees are constructed as follows:

1. Repeat below for $i = 0, 1, \ldots, k-2$.
2. Create initial trees T_i^* which consists of the root node and its $k-i$ children.
3. Execute from second to seventh step of previous algorithm.

The encoding algorithm with multiple parse trees is as follows:

1. Construct the multiple parse trees.
2. Let $T \leftarrow T_0$, n be the root of T_0.
3. Repeat below until the end of the input text.
4. Let c be the next symbol of the input text.
5. If there if no child of n in which we can traverse by symbol c, then output the codeword of n, $T \leftarrow T_{d(n)}$, and let n be the root of T. Otherwise, let n be the child of n labeled by c.

For example, Let an input text be $S = aabaabaccab$, and consider to encode S with the parsing trees thus. Then, S is split into blocks, $aa \cdot baa \cdot bac \cdot ca \cdot b$. Therefore, we obtain the output binary sequence of length 15 bits as $001 \cdot 001 \cdot 011 \cdot 111 \cdot 101$.

3 Virtual Multiple AIVF tree

In this section, we explain our idea and present an efficient algorithm for AIVF codes.

In AIVF parse trees, we can observe that many nodes in T_i are identical with those in T_{i+1}. More precisely, T_{i+1} completely covers the nodes in T_i except for the leftmost subtree under the node corresponding with a_{i+1}. First, we explain this relationship.

Let $S_j^{(i)}$ be the subtree of T_i that consists of all the nodes under the node corresponding with a_j, which is a direct child of the root. Then, we have the following theorem.

Theorem 1 *$S_{i+j}^{(i+1)}$ completely covers $S_{i+j}^{(i)}$ for any integers i $(0 \leq i \leq k-3)$ and j $(2 \leq j \leq k-i)$.*

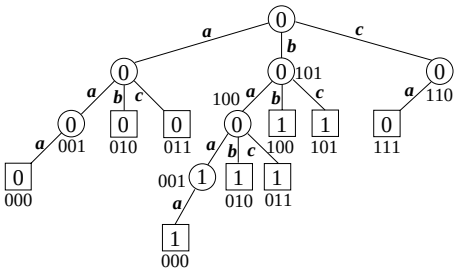

Figure 4: An example of VMA tree

This is an integrated tree obtained by multiplexing the tree in Fig. 2 and in Fig. 3. The number written in each node indicates the value of $\mathrm{Tn}(n)$.

Proof. We prove the theorem by contradiction.

For an integer i $(0 \leq i \leq k-2)$, let T_i^{∞} be the tree of infinite depth in which the root has children according with $a_{i+1}, \ldots, a_k$ and all the other internal nodes have just k children.

Note that the root of the multiple parse tree T_i has children according with a_j $(j = i+1, \ldots, k)$. That is, T_i includes $S_j^{(i)}$ $(j = i+1, \ldots, k)$. Also note that each T_i is an optimal in the sense of maximizing the average block length. That is, for any i $(0 \leq i \leq k-3)$ and j $(2 \leq j \leq k-i)$, we can not increase the average block length any more by exchanging any node in $S_{i+j}^{(i)}$ for a node included in T_i^{∞} but not in T_i.

Here, we assume that $S_{i+j}^{(i+1)}$ does not cover $S_{i+j}^{(i)}$ completely. Then, there exists a node which is in $S_{i+j}^{(i)}$ but not in $S_{i+j}^{(i+1)}$. Let n be one of such the nodes. Since T_i is a parse tree which maximizes the average block length, we can increase it by exchanging a node in $S_{i+j}^{(i+1)}$ but not in $S_{i+j}^{(i)}$ for n. However, this contradicts that T_{i+1} maximizes the average block length. Therefore, $S_{i+j}^{(i+1)}$ completely covers $S_{i+j}^{(i)}$. □

From the above theorem, we can observe that we can multiplex the set of trees and integrate into a single tree in a simple way. We call the integrated tree as *Virtual Multiple AIVF tree* (VMA tree for short). To simulate the encoding and the decoding of the original AIVF codes by using VMA tree, we need to tell which parse tree we are currently traversing while processing. Thus, we mark each node in VMA tree in order to tell which trees the node belongs to. Note that a node can belong to several parse trees. We can realize that by holding the least i such that n belongs to T_i. Denoting this mark by $\mathrm{Tn}(n)$, we have that $\mathrm{Tn}(n) = \min_i\{i \mid 0 \leq i \leq k-2, n \text{ belongs to } T_i.\}$. For example, integrating two parse trees shown in Fig. 2 and Fig. 3, we can obtain a parse tree shown in Fig. 4.

To encode with VMA tree, we need to modify the previous encoding Algorithm. The reason is that, even if there is a child in the VMA tree for the next traverse, the encoding fails when there is not the child in T_i. Therefore, we compare $\mathrm{Tn}(n)$ and the number i of currently traversing tree T_i. If i is less than $\mathrm{Tn}(n)$, then there is not a proper node in T_i, and return to the root. The encoding algorithm is shown in Algorithm 1. In Algorithm 1, we denote by $w_i(n)$ the codeword assigned to n in T_i.

Algorithm 1 Encoding algorithm with VMA tree.

```
 1: n ← The root of T
 2: i ← 0
 3: while Not end of the input text do
 4:    c ← The next symbol of the input text
 5:    if There exists a child of n labeled by c then
 6:       n' ← The child of n labeled by c
 7:       if Tn(n') ≤ i then
 8:          n ← n'
 9:       else
10:          Goto the line 13
11:       end if
12:    else
13:       Output w_i(n)
14:       i ← d(n)
15:       n ← The root of T
16:       n ← The child of n labeled by c
17:    end if
18: end while
```

The algorithm of constructing VMA tree is shown in Algorithm 2. We denote by S_i the subtree which consists of all nodes under the node corresponding with a_i. We define $\#\mathcal{N}(S_0) = M$ for convenience. We define the function $\mathrm{cod}(a_j) = j$, and denote by $\mathrm{first}(n)$ the first symbol of the label sequence on the path from the root to n. That is, if $\mathrm{first}(n) = a_j$, $\mathrm{cod}(\mathrm{first}(n)) = j$. We also denote by m the number of nodes having codeword in Algorithm 2.

3.1 The number of nodes in VMA tree

Here we discuss about the number of nodes in VMA tree.

Let M be the number of codewords, i.e., $M = 2^\ell$ for the codeword of length ℓ. Recall that, for each T_i $(0 \le i \le k-3)$ in the original AIVF tree, $S^{(i)}_{i+j}$ $(2 \le j \le k-i)$ is completely covered from Theorem 1. This implies that the VMA tree is identical to the union of $S^{(0)}_1, S^{(1)}_2, \ldots, S^{(k-3)}_{k-2}$, and T^{k-2}. Then, for each i $(0 \le i \le k-3)$, $S^{(i)}_{i+1}$ includes at most $M-(k-i-1)$ nodes because $S^{(i)}_j$ for any i, j has at least one node. Therefore, the total number of nodes in VMA tree is as follows:

$$\begin{aligned}\sum_{i=0}^{k-3}(M-k+i+1)+M &= Mk-\frac{1}{2}k^2+\frac{1}{2}k-M+1\\ &= O(Mk-k^2).\end{aligned}$$

Intuitively, the number of reduction is $\Omega(k^2)$ because the total number of nodes in the original AIVF trees is $O(Mk)$. Although we proved that, we omit the detail derivation. We just show the result below. Let N and N_v be the number of nodes in a AIVF tree and that in the equivalent VMA tree, respectively. Then, we have:

$$N - N_{\mathsf{V}} \ge \left(\frac{k^2}{2}-\frac{k}{2}-1\right)+\left(\sum_{i=0}^{k-2} m_i^{\mathsf{C}} - m_{\mathsf{V}}^{\mathsf{C}}\right),$$

Algorithm 2 Constructing a VMA tree

$T \leftarrow$ The tree with root and k children of it;
Label the jth edge of T by a_j;
for $l = 0$ to $k-2$ **do**
  $m \leftarrow \#\mathcal{N}(S_l)$; $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T)} \Pr(n)$;
  while $k - d(\hat{n}) - 1 \leq m$ **do**
    $S_1 \leftarrow$ The average block length assuming that we call Complete();
    $S_2 \leftarrow$ The average block length assuming that we call FindOptPos() $k - d(\hat{n}) - 1$ times;
    if $S_1 \geq S_2$ **then**
      Call Complete();
    else
      Call FindOptPos $k - d(\hat{n}) - 1$ times;
    end if
    $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T)} \Pr(n)$; $m \leftarrow m - k + d(\hat{n}) + 1$;
  end while
  Call FindOptPos m times ;
  $i \leftarrow 0$;
  for all $n \in \mathcal{N}(D)$ **do**
    $w_l(n) \leftarrow i$; $i \leftarrow i + 1$;
  end for
  $S \leftarrow$ The subtree corresponding the node traversed from the root by a_{l+1};
  $R \leftarrow T$ except for S; Add S to T_{V}; $T \leftarrow R$;
end for
Label the jth edge of each node by a_j $(j = 1, \ldots, k)$.

Procedure Complete()
  $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T)} \Pr(n)$;
  Add $k - d(\hat{n})$ children n_j, $j = d(\hat{n}) + 1, \ldots, k$ to $\hat{n}$;
  $\mathrm{Tn}(n_j) \leftarrow \mathrm{cod}(\mathrm{first}(n_j)) - 1$;

Procedure FindOptPos()
  $\tilde{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T)} \Pr(n) \Pr(a_{d(n)+1})$;
  Add the $d(\tilde{n}) + 1$th child n of $\tilde{n}$;
  $\mathrm{Tn}(n) \leftarrow \mathrm{cod}(\mathrm{first}(n)) - 1$;

where m_i^{C} and $m_{\mathsf{V}}^{\mathsf{C}}$ are the number of complete internal nodes in T_i of the AIVF trees and that in the VMA tree, respectively. Thus, the number of nodes reduced is $\Omega(k^2)$ because the second parenthesis $\left(\sum_{i=0}^{k-2} m_i^{\mathsf{C}} - m_{\mathsf{V}}^{\mathsf{C}}\right)$ is equal to or greater than 0.

4 Experimental Results

We have implemented both AIVF code and our proposed method. We abbreviate those programs as AIVF and VMA, respectively. All programs we used are written in C++ and compiled by g++ of GNU, version 4.3. We ran our experiments on an Intel Pentium 4(R) processor of 3.00GHz with 2GB of RAM running on Debian GNU/Linux 5.0. We used bible.txt as a natural language text, which is the file of "The King James version of the bible" selected from "the Canterbury corpus[2]." Its size is $4,047,392$ bytes and the alphabet size is 63.

[2]http://corpus.canterbury.ac.nz/descriptions/

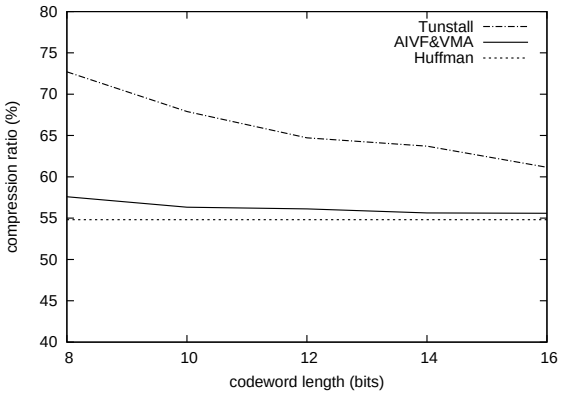

Figure 5: Compression ratio against codeword length.

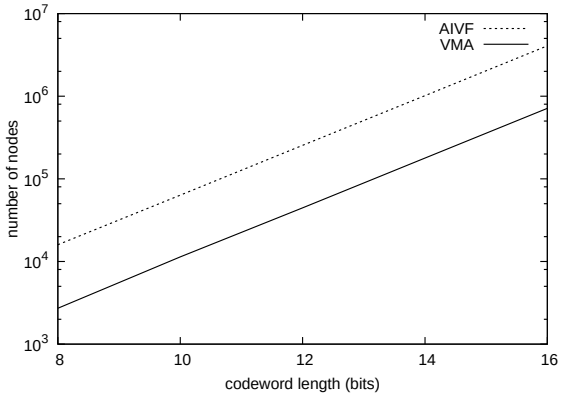

Figure 6: Total number of nodes against codeword length.

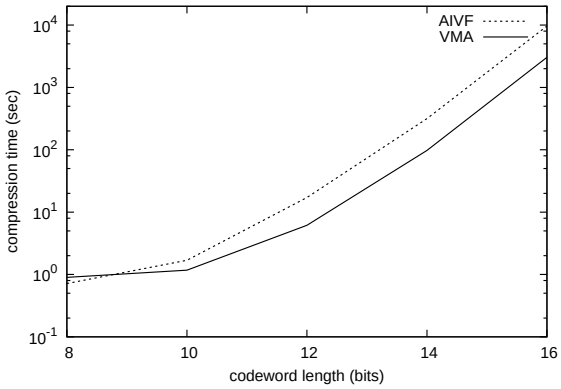

Figure 7: Compression time against codeword length.

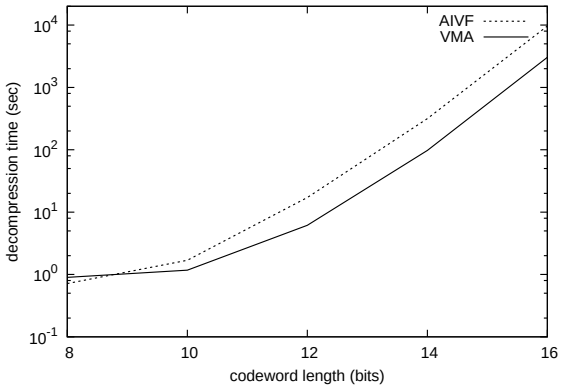

Figure 8: Decompression time against codeword length.

At first, we show the compression ratio of AIVF codes for reference in Fig. 5. We measured (compressed file size)/(original file size) as the compression ratio, and added Tunstall code and Huffman code just for reference. As shown in Fig. 5, we can see that AIVF(VMA) is almost competitive with Huffman code. Note that AIVF and VMA output the same codes, that is, those have the same compression ratio.

Next, we compared the number of nodes and the encoding/decoding times against the length of codewords. In this comparison, we only used bible.txt. The results are shown in Figures 6, 7, and 8. We measured them by the CPU times calculated by the time command of Linux. Note that the encoding times include the time computing the compression model, and that the graphs are semilogarithmic. As shown in Fig. 7 and Fig. 8, VMA is approximately over three times faster than AIVF when the codeword length is greater than 10.

Last, we compared the number of nodes when we fixed $l = 14$ and used randomly generated texts for various alphabet sizes. Figure 9 shows the result. We can see that the number of nodes in VMA is drastically reduced as the alphabet size becomes larger. Although we also compared for natural language corpora, we omit the results because they were almost identical with the result shown in Fig. 9.

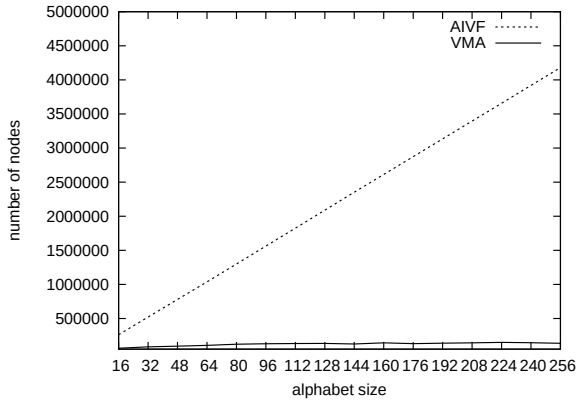

Figure 9: Number of nodes against the alphabet size

5 Conclusions

We presented an efficient algorithm for AIVF codes, which integrates the multiple parse trees of an AIVF code into one, named a VMA tree, and simulates the encoding process and the decoding process on it. We also presented that we can reduce the total number of nodes by $\Omega(k^2)$, that is, the number of nodes is a VMA tree is $O(2^\ell k - k^2)$. This indicates that we can reduce the time and space to construct the parse tree by $\Omega(k^2)$. The experimental results showed that our method runs much faster than the original one in fact.

Although the methods in [KS09, Kid09] have better compression ratios rather than AIVF codes, they need to construct suffix trees [CR02] for constructing a parse tree, and thus, they take much time for doing it. Therefore, from the viewpoint of speeding up compressed pattern matching, AIVF code with VMA tree is a strong candidate as well, because we usually need to construct a parse tree for doing compressed pattern matching on VF codes.

References

[CR02] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[Kid09] T Kida. Suffix tree based VF-coding for compressed pattern matching. In *Proc. of Data Compression Conference 2009(DCC2009)*, page 449, Mar. 2009.

[KS09] Shmuel T. Klein and Dana Shapira. Improved variable-to-fixed length codes. In *SPIRE '08: Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, pages 39–50, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sav98] Serap A. Savari. Variable-to-fixed length codes for predictable sources. In *Proc. of DCC98*, pages 481–490, 1998.

[Tun67] B. P. Tunstall. *Synthesis of noiseless compression codes.* PhD thesis, Georgia Inst. Technol., Atlanta, GA, 1967.

[YY01] Hirosuke Yamamoto and Hidetoshi Yokoo. Average-sense optimality and competitive optimality for almost instantaneous VF codes. *IEEE Trans. on Information Theory*, 47(6):2174–2184, Sep. 2001.