Journal of Information Processing Vol.20 No.1 238–249 (Jan. 2012)

[DOI: 10.2197/ipsjjip.vol.20.238]

Regular Paper

Improving Parse Trees for Efficient Variable-to-Fixed Length Codes

Satoshi Yoshida[1,a)] Takashi Uemura[1,†1] Takuya Kida[1]
Tatsuya Asai[2] Seishi Okamoto[2]

Received: April 11, 2011, Accepted: September 12, 2011

Abstract: We address the problem of improving variable-length-to-fixed-length codes (VF codes). A VF code that we deal here with is an encoding scheme that parses an input text into variable length substrings and then assigns a fixed length codeword to each parsed substring. VF codes have favourable properties for fast decoding and fast compressed pattern matching, but they are worse in compression ratio than the latest compression methods. The compression ratio of a VF code depends on the parse tree used as a dictionary. To gain a better compression ratio we present several improvement methods for constructing parse trees. All of them are heuristical solutions since it is intractable to construct the optimal parse tree. We compared our methods with the previous VF codes, and showed experimentally that their compression ratios reach to the level of state-of-the-art compression methods.

Keywords: lossless data compression, tunstall code, STVF code, suffix tree, almost instantaneous coding

1. Introduction

Data compression is one of the most fundamental operations in text processing, whose aim is to reduce the storage space usage and the cost of transfer for massive amounts of data. The compression ratio is usually considered as the most important factor in this research area. Many compression methods have been proposed so far (see Refs. [21] and [24]). Among them, the Ziv-Lempel family [31], [32] is one of the most popular in practice because of their good compression ratios and fast processing. Since a significant and global increase in the use of the Internet and e-mails in recent years has caused the explosion of unformatted text data, data compression becomes more and more important as the foundation of textual databases for such enormous text data.

For large-scale textual databases, however, such a state-of-the-art compression method is unsuitable. Although well-known compression tools such as gzip and bzip2 can compress text data extremely, they encode with variable-length codewords and the encoded texts are highly complicated. It is difficult for users to search a part of data without decompressing, and thus it is difficult to reuse the compressed data. Therefore, *fast searchable compression methods* are strongly desired.

From the viewpoint of speeding up pattern matching on compressed texts, *variable-length-to-fixed-length codes* (VF codes for short) have been reevaluated recently [11], [15]. A VF code is a coding scheme that parses an input text into a consecutive sequence of substrings (called blocks) with a dictionary tree, which is called a parse tree, and then assigns a fixed length codeword to each substring; such a codeword enables us to touch any parsed block randomly without concerning codeword boundaries.

Several promising VF codes have been proposed so far. Maruyama et al. [18] proposed an excellent compression method, which is a variation of grammar-based compressions. They proposed Σ-sensitive grammar for effective grammar transform. In their practical implementation, which we call BPEX[*1], the method can also be viewed as a VF code since an encoded text is represented as a sequence of grammar symbols, which are represented by fixed length codewords of length 8-bits; this means the number of grammar symbols is bounded by 256. Although BPEX achieves a good compression ratio comparable to gzip, its compression speed is slow.

VF codes based on suffix trees [10] are proposed independently by Klein et al. [15] and Kida [11]. In their scheme, a frequency-base-pruned suffix tree is used as a parse tree. An input text is scanned once at first to construct the suffix tree, and then the text is scanned again and translated into a sequence of codewords. Although they are faster than BPEX in compression speed, their compression ratios are not better than that of BPEX. A VF code that achieves both fast compression/decompression and high compression ratio is desired.

The compression ratio of a VF code depends on the parse tree, namely, the most important issue is which substrings we should enter into the parse tree. Let Σ be an alphabet and k be a positive integer. Consider a text $T := t_1 t_2 \cdots t_n$ to be encoded by k-bit fixed length codes, where $t_i \in \Sigma$. The aim here is to make an efficient dictionary D, which consists of different substrings of T, such that T can be parsed uniquely into a sequence of entries of D. Each entry of D is assigned a codeword of length k bits, thus

[1] Hokkaido University, Sapporo, Hokkaido 060–0814, Japan
[2] Fujitsu Laboratories Ltd., Kawasaki, Kanagawa, 211–8588, Japan
[†1] Presently with Yahoo Japan Corporation
[a)] syoshid@ist.hokudai.ac.jp

[*1] This name comes from the program implemented by Maruyama.

© 2012 Information Processing Society of Japan

the number of entries in D is less than or equal to 2^k. If the text T is parsed with D into a sequence of m blocks, $T := c_1 c_2 \cdots c_m$ ($c_i \in D$), the size of the encoded text becomes km bits in addition to the size of D. Therefore, we want to make a dictionary such that

$$km + \sum_{c \in D} |c|$$

is minimized under $|D| \leq 2^k$. However, this problem is quite hard as Klein and Shapira stated in Ref. [15]:

> Choosing an optimal set of substrings might be intractable, since even if the strings are restricted to be the prefixes or suffixes of words in the text, the problem of finding the set is NP-complete [8], and other similar problems of devising a code have also been shown to be NP-complete in Refs. [6], [7], [14]. A natural approach is thus to suggest heuristical solutions and compare their efficiencies.

Our concern for this problem is how to construct a parse tree that approximates the optimal tree well. In most of the VF codes, a frequency of each substring of T is often used as a clue for approximation, since it is related to the number of occurrences of the substring in a sequence of parsed blocks. This gives a chicken and egg problem as Klein and Shapira also stated in Ref. [15]. To construct a better parse tree, we need the frequency of each block. To know the frequency of each block, we must determine the partition of T. However, we need a parse tree before determining the partition.

In this paper we present several methods to improve the compression ratio of VF codes. First, we use a pruned suffix tree as the parse tree in order to catch the context in the input text. We call this method *STVF codes*. This work has already been partially presented in Refs. [11] and [12]. The experimental results show that the STVF code achieves the compression ratio of about $42\% - 50\%$ for natural language texts. Next, we extend the STVF code so that it can allow to assign codewords to incomplete internal nodes in the parse tree in order to improve the compression ratio, while only leaves are assigned the codewords in the original STVF code. We also employ a method that chooses nodes one by one from the suffix tree in descending order of their frequencies. In this method, the encoding process is like *almost instantaneous VF codes* (*AIVF codes* for short) proposed by Yamamoto et al. [28]. We call this method *almost instantaneous STVF codes* (*AISTVF codes* for short). We show experimentally that the AISTVF code improves the compression ratio more than 18% for natural language texts in comparison with the STVF code.

Moreover, we present a way of training the parse tree by compressing the input text and modifying the parse tree repeatedly. We also discuss a method that uses parts of the input text for training in order to reduce the training time. The training method improves the compression ratio of VF codes rapidly to the level of state-of-the-art compression methods. This work has already been partially presented in Ref. [27].

The rest of this paper is organized as follows. In Section 2, we give a brief review on related works. In Section 3, we make a brief sketch of the Tunstall code and suffix trees with some basic notations. In Section 4, we introduce the original STVF code and present its improvement by the almost instantaneous coding strategy [28]. Moreover, we show our experimental results of the comparison of our methods with the Tunstall code and BPEX. In Section 5, we present the method of training parse trees and show several experimental results. Finally, we conclude in Section 6.

2. Related Works

Maruyama et al. [18] presented an excellent compression method named as BPEX for compressed pattern matching, which is a variation of Byte-Pair-Encoding [9] (BPE for short) that is a kind of VF codes. In BPE scheme, the encoding procedure scans the input text many times and repeats the conversion of a frequent 2-gram (pair of bytes) into an unused bytecode. Moreover, BPEX employs the technique of switching dictionaries according to the context.

Larsson et al. [17] conceived RE-PAIR. Nevill-Manning et al. [20] advocated SEQUITUR. They are kinds of the grammar-based compressions, which construct a grammar at first that derives uniquely the input text, and then encode the grammar with an entropy encoding, such as the Huffman codes and the arithmetic encoding.

Brisaboa et al. [5] proposed End-Tagged Dense Code (ETDC), and Brisaboa et al. [4] did (S, C)-Dense Code (SCDC). ETDC and SCDC are word based compressions that encode each word in the input text into a binary codeword. Brisaboa et al. [4] developed Dynamic ETDC (DETDC) and Dynamic SCDC (DSCDC), which are adaptive approaches of ETDC and SCDC, respectively. Moreover, Brisaboa et al. [2] propounded Dynamic Lightweight ETDC (DLETDC) and Dynamic Lightweight SCDC (DLSCDC), which are easily-decodable versions of DETDC and DSCDC, respectively.

Brisaboa et al. [3] recently presented *v2vdc*. It uses suffix arrays to catch efficiently the context in the input text. Klein et al. [16] devised a method based on the dense coding using the Fibonacci codes [1] for text compression.

3. Preliminaries

3.1 Basic Definitions

Let Σ be a finite alphabet. We denote the set of all strings over Σ by Σ^*. The *length* of a string $T := t_1 t_2 \cdots t_n \in \Sigma^*$ ($t_i \in \Sigma$ for any i) is denoted by $|T|$. Therefore, we have $|T| = n$. The string of length zero, denoted by ε, is called the *empty string*. The set of all strings on Σ except the empty string ε is denoted by Σ^+. Let T be a string in Σ^*. Then, strings x, y and z in Σ^* are called a *prefix*, *substring*, and *suffix* of T respectively if the input text T is represented by the concatenation of three strings x, y, and z, that is, $T = xyz$.

3.2 Variable-length-to-fixed-length Codes

A VF code is a source coding that parses an input string into a consecutive sequence of variable length substrings and then assigns a fixed length codeword to each substring. We call the parsed substring a *block*. There are many variations on how they parse the input string, what kind of data structures they use as

© 2012 Information Processing Society of Japan

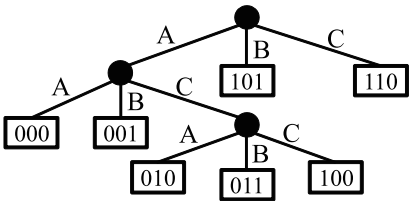


Fig. 1 An example of a parse tree. The squares represent leaves, where codewords are assigned. The circles represent internal nodes.

a dictionary, and how they assign codewords. Among them, the method that uses a tree structure, called a parse tree, is the most fundamental and common.

We explain the encoding and the decoding procedure of VF codes. Consider that we encode an input text $T \in \Sigma^*$ by a VF code of length k-bits codewords. Assume that a parse tree $\mathcal{T}$ that has ℓ leaves is given, and that each leaf in $\mathcal{T}$ is numbered by a k-bit integer, that is, we have $\ell \leq 2^k$. Then, we can parse and encode T with $\mathcal{T}$ as follows:

(1) Start a traversal at the root of $\mathcal{T}$.
(2) Read a symbol one by one from T, and traverse the parse tree $\mathcal{T}$ by the symbol. If the traversal reaches to a leaf, then output the codeword assigned at the leaf before getting back to the root.
(3) Repeat Step 2 till T ends.

For example, given the text $T :=$ AAABBACB and the parse tree of **Fig. 1**, the encoded sequence is 000/001/101/011. Codeword 011, for the running example, represents block ACB. A decoding process of a VF code is quite simple. First we restore the parse tree. Then we replace each codeword with the corresponding block as referring to the restored parse tree.

The Tunstall code [26] is known to be an optimal VF code (see also Ref. [23]) for a memory-less information source. Its average code length per symbol comes asymptotically close to the entropy of the input source when the codeword length goes to infinity. It uses a parse tree called the Tunstall tree, which is the optimal tree in the sense of maximizing the average block length. The Tunstall tree is an ordered complete $|\Sigma|$-ary tree in which each edge from the same node is labeled with a distinct symbol in the alphabet Σ. Let $\Pr(a)$ be an occurrence probability for source symbol $a \in \Sigma$. The probability of string $x_\mu \in \Sigma^+$, which is represented by the path from the root to leaf μ, is $\Pr(x_\mu) = \prod_{\eta \in \xi} \Pr(\eta)$ for the memoryless information source, where ξ is the label sequence on the path from the root to μ (hereafter, we identify a node in $\mathcal{T}$ and a string represented by the node if no confusion occurs). Then, the Tunstall tree $\mathcal{T}^*$ is constructed as follows:

(1) Initialize $\mathcal{T}^*$ as the ordered $|\Sigma|$-ary tree whose depth is 1, which consists of $|\Sigma| + 1$ nodes.
(2) Repeat the following while the number of leaves in $\mathcal{T}^*$ is less than or equal to 2^k
  (a) Select a leaf v that has the maximum probability among all leaves in $\mathcal{T}^*$.
  (b) Make v be an internal node by adding $|\Sigma|$ children onto v.

Let m be the number of internal nodes in $\mathcal{T}^*$. Since the number of leaves in $\mathcal{T}^*$ equals to $m(|\Sigma|-1)+1$, which is less than or equal to 2^k. The above method creates the largest tree which satisfies the condition. Hence, $m = \lfloor (2^k - 1)/(|\Sigma| - 1) \rfloor$ holds. For an information source with memory, there have been proposed several coding methods that are based on the Tunstall code [22], [25].

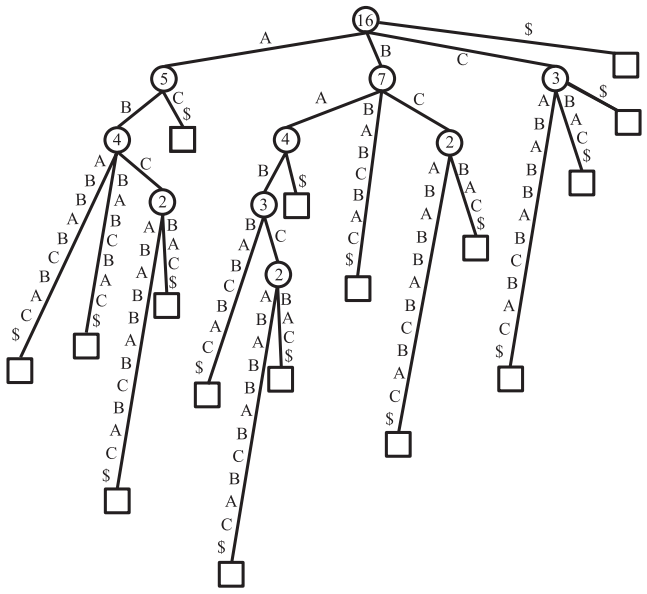


Fig. 2 Suffix tree for BABCABABBABCBAC$. The squares represent leaves. The circles represent internal nodes and the numbers in the circles are their frequencies.

3.3 Suffix Trees

The *suffix tree* of a given text T is a compacted trie which represents all the suffixes of T. Formally, $ST(T\$)$ is defined as follows:

(1) Each internal node, except the root of $ST(T\$)$, has at least two children.
(2) Each edge is labeled by a non-empty substring of T. For a node v, we denote the label of the incoming edge of v by $label(v)$.
(3) For any internal node u, any labels of outgoing edges start with different characters each other.
(4) Let the *representing string* $str(v)$ of a node v in $ST(T\$)$ be the string obtained by concatenating the labels of the edges in the path from the root to v[*2]. Then, each leaf of $ST(T\$)$ corresponds one-to-one with each suffix of T, where \$ is a special symbol not in Σ.

For example, the suffix tree of a string BABCABABBABCBAC\$ is shown in **Fig. 2**. For a node v in $ST(T\$)$ and a symbol $c \in \Sigma$, the function $child(v, c)$ and $f(v)$ return the child of v whose label of the incoming edge starts with c and the number of occurrences of $str(v)$ in T, the frequency of a node v, respectively. An internal node u in the parse tree is said to be *complete* if the parse tree contains all the children of u in $ST(T)$, otherwise u is said to be *incomplete*. Since the frequency of node v is equal to the number of leaves in the subtree rooted at v, the computation of the frequencies of all nodes is done in a post-order traversal. Note that the suffix tree for T is constructed in linear time in the length of T.

For more details, please refer to Ref. [10].

[*2] The representing string of the root node is the empty string, that is, $str(root) = \varepsilon$.

© 2012 Information Processing Society of Japan

4. STVF Codes

4.1 VF Code by Pruned Suffix Tree

In this section we introduce a VF coding that uses a pruned suffix tree for a parse tree, which is named as the STVF code and firstly (partially) presented in Ref. [11].

The Tunstall code does not achieve a good compression ratio for an information source with memory because it assumes that the information source is memoryless. As stated in the previous section, a suffix tree stores all substrings of the given text; moreover the frequency of any substring can be easily obtained. This suggests that the suffix tree can be a good base of the parse tree for the given text. For a given text T, the deepest leaf, which is the leaf v such that $str(v)$ is the longest among all leaves, represents T itself. Therefore the whole $ST(T)$ can not be used as a parse tree. The idea of our new VF code is to prune deeper nodes in $ST(T)$ and make it a compact parse tree.

We denote by $ST_L(T)$ a pruned suffix tree such that the number of leaves equal to L by pruning. Note that a pruned suffix tree includes all nodes whose depth is 1 that are also included in the original $ST(T)$, and that it includes any symbols which occur in T. Now consider to encode T by codewords of length ℓ. As the same as the Tunstall code, the formula $L \leq 2^\ell$ must be satisfied. The procedure to parse and encode T with $ST_L(T)$ is also the same way as the Tunstall code.

The simplest strategy of pruning is to search $ST(T)$ by breadth-first-search from the root, and select the shallowest nodes till the number of leaves in a pruned suffix tree is up to L. A more sophisticated way is to select the nodes so that the frequencies of the leaves in $ST_L(T)$ become nearly uniform. Namely, select the nodes in the order of their frequencies from the root. Our pruning procedure is as follows:

(1) Construct a suffix tree $ST(T)$.
(2) Let a tree $ST_{k+1}(T)$ which consists of the root of $ST(T)$ and its direct children, be the first parse tree candidate $\mathcal{T}_1$.
(3) Select a node v such that the number of children of v in the sense of $ST(T)$ is the largest among all leaves in $\mathcal{T}_i = ST_{L_i}(T)$ where L_i is the number of leaves in $\mathcal{T}_i$. Here let C_v be the number of direct children of v.
(4) Add all children of v to $\mathcal{T}_i$ as leaves if $L_i + C_v - 1 \leq 2^\ell$, and let it be $\mathcal{T}_{i+1}$ for a new candidate of a parse tree. If a child u of v is a leaf in $ST(T)$, delete the label on the edge from v to u except for the first character of the label.
(5) Repeat Steps 2 and 3 till we can not select any node.

Figure 3 is the parse tree construction algorithm of the STVF code, and **Fig. 4** shows the parse tree $ST_8(T)$ for $T :=$ BABCABABBABCBAC constructed by the algorithm. The algorithm first constructs the suffix tree $ST(T)$ for an input text T. Next, for each step of the outer loop (from Lines 4 to 13), the most frequent node v among the leaves in the temporal parse tree $\mathcal{T}_i$ is selected, and then all the children of v in $ST(T)$ are added to $\mathcal{T}_i$. The algorithm removes the label of an incoming edge of a child of v except for the first character of the label if the child is a leaf in $ST(T)$. After the above pruning steps, the algorithm assigns codewords to all the leaves in a left-to-right manner. The first four iterations of the constructing process for the running example is shown in **Fig. 5**. This construction strategy is similar to that of the Tunstall code.

Algorithm STVF(T, k):
Input: A text T and the codeword length k.
Output: The parse tree $ST_{L'}(T)$.

1: Construct the suffix tree $ST(T)$ of T;
2: Construct the initial tree $\mathcal{T}$ which only contains the root of $ST(T)$;
3: $U \leftarrow \{root\}$; //set of nodes that will be assigned codewords
4: **while** $|U| < 2^k$ **do**
5: $v \leftarrow \text{argmax}_{v \in U} f(v)$;
6: $U \leftarrow U \setminus \{v\}$;
7: **for each** child w of v **do**
8: $U \leftarrow U \cup \{w\}$;
9: **if** w is a leaf of $ST(T)$ **then**
10: remove $lavel(w)$ except for the first character of it;
11: add w to $\mathcal{T}$;
12: **end for**
13: **end while**
14: assign codewords to the elements in U;
15: **return** $\mathcal{T}$ as $ST_{|U|}(T)$;

Fig. 3 Algorithm of constructing a parse tree of STVF code.

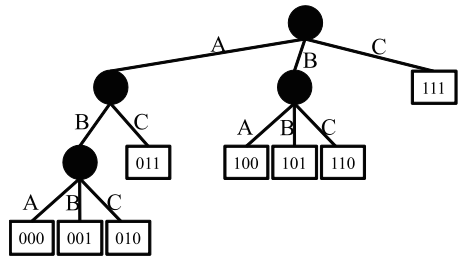


Fig. 4 Parse tree of the STVF code for BABCABABBABCBAC. The squares and the circles indicate leaves and internal nodes, respectively. The numbers in squares are assigned codewords.

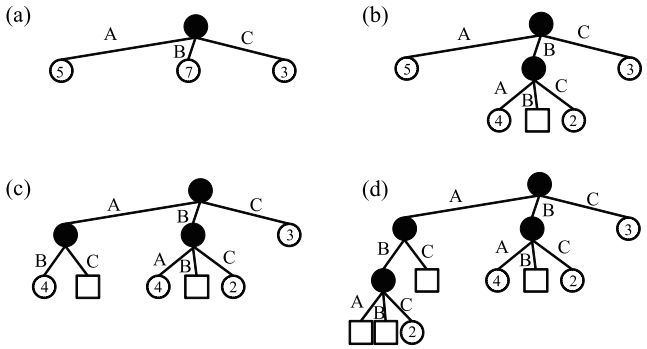


Fig. 5 The first four iterations of the construction process of the parse tree. The black circles indicate internal nodes. Only leaves are assigned codewords.

For the parse tree $ST_{L'}(T)(L' \leq L)$ obtained by the algorithm, we have the following lemma.

Lemma 1. *For a given text T, we can uniquely parse T by using the pruned suffix tree $ST_{L'}(T)$.*

Proof. Let D be a set of strings which is entered into the pruned suffix tree $ST_{L'}(T)$, and call D as a *dictionary*. From the pruning

© 2012 Information Processing Society of Japan

procedure each leaf in $ST_{L'}(T)$ corresponds one-to-one to each string entered in D. Therefore, all the strings in D satisfy the prefix condition since only leaves are assigned the codewords, that is, for any string $s \in D$, there exists no string $t \in D$ such that $t \neq s$ and s is a prefix of t. Hence, we can uniquely parse the input text T. □

Once the parse tree is constructed, the encoding and decoding procedures are simple: they are shown in **Fig. 6** and **Fig. 7**, respectively. For the running example in Fig. 4, the text is parsed into seven substrings as BA/BC/ABA/BB/ABC/BA/C, and encoded to 100/110/000/101/010/100/111. We must store the parse tree together with the sequence of encoded blocks. We divide the parse tree into two components: the tree structure and the labels on it. The tree structure is encoded by balanced parentheses [19]. Thus the encoded size for the tree of M nodes is $2M$ bits. For the labels, we store them by a simple way: enumerate pairs of the label length and the label string and then attach to the encoded tree structure. Assuming that each label length is smaller than 256, which can be represented by one byte, the set of labels can be stored by $\sum_{l \in L}(|l| + 1)$ bytes, where L is the set of labels.

The following lemma and theorem suggest the performance of the STVF code.

Lemma 2. *The parse tree constructed by the Algorithm STVF of* Fig. 3 *is equivalent to the Tunstall tree for a sufficiently long string of arbitrary memoryless information sources.*

Proof. For a node p in the suffix tree and a character c, $f(p) \cdot Pr(c) = f(p \cdot c)$ holds with a sufficiently long string because we assume that the information source is memoryless. Therefore, we obtain the occurrence probability of the representing string of each node by dividing the frequency of each node by that of the root node. Both of the parse tree construction algorithms of the STVF code and the Tunstall code select the leaf that has the maximum probability to add all its children. Therefore, both of the algorithms select the same node. Hence, the STVF tree and the Tunstall tree are equivalent for arbitrary memoryless information sources. □

Theorem 1. *The number of codewords output by the STVF code is the same as the one output by the Tunstall code for arbitrary memoryless information source.*

Proof. The parse tree of the STVF code is equivalent to the Tunstall tree for memoryless information sources from Lemma 2. Therefore, the number of codewords output by the STVF code and the one output by the Tunstall code are the same for arbitrary memoryless information sources. □

4.2 Improving the Compression Ratio by Almost Instantaneous Coding

Next we present an improved version of the STVF code stated in the above. In the STVF code, unused codewords of length k exist if there does not exist an integer m satisfying $m(|\Sigma|-1)+1 = 2^k$. This suggests that we can encode the input text with fewer codewords 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 acquire much better compression ratios if we remove low-frequency leaves and extend useful edges.

We introduce the algorithm for constructing a parse tree as in **Fig. 8**. The basic idea of the algorithm is to choose the most fre-

Algorithm encode($T, \mathcal{T}$):
Input: A text T and a parse tree $\mathcal{T}$.
Output: An encoded sequence of codewords.

```
1: i ← 0;
2: while i < |T|
3:     v ← root;
4:     while v is a internal node of T do
5:         v ← the node that represents str(v) · T[i];
6:         i ← i + 1;
7:     end while
8:     output the codeword assigned to v;
9: end while
```

Fig. 6 Encoding algorithm of the STVF code.

Algorithm decode($\mathcal{T}, C$):
Input: A parse tree $\mathcal{T}$ and a sequence C of codewords.
Output: Decoded text T.

```
1: for each i ∈ {0, ..., |C| − 1} do
2:     v ← the node such that code(v) = C[i];
3:     output str(v);
4: end for
```

Fig. 7 Decoding algorithm of the STVF code.

Algorithm ImprovedSTVF(T, k):
Input: A text T and the length k of codewords.
Output: A parse tree $\mathcal{T}$ for T.

```
1: Construct the suffix tree ST(T) of T;
2: Construct the parse tree T which only contains the root of ST(T);
3: U ← ∅, V ← {v | v is a child of the root of ST(T)};
4: for each child v of the root of ST(T) do
5:     add v to T;
6:     if v corresponds to a leaf in ST(T) then
7:         remove label(v) except for the first character of it;
8:     end if
9:     U ← U ∪ {v};
10:    V ← (V \ {v}) ∪ {w | w is a child of v};
11: end for
12: while |U| < 2^k do
13:    v ← argmax_{v∈V} f(v);
14:    add v to T;
15:    U ← U ∪ {v};
16:    V ← (V \ {v}) ∪ {w | w is a child of v};
17:    p ← v's parent;
18:    if #{w ∈ V | w is a child of p} = 1 then
19:        w ← p's just one child remaining in V;
20:        U ← (U \ {p}) ∪ {w};
21:        V ← (V \ {w}) ∪ {x | x is a child of w};
22:    end if
23: end while
24: assign codewords to the elements in U;
25: return T;
```

Fig. 8 Improved construction algorithm for parse trees.

© 2012 Information Processing Society of Japan

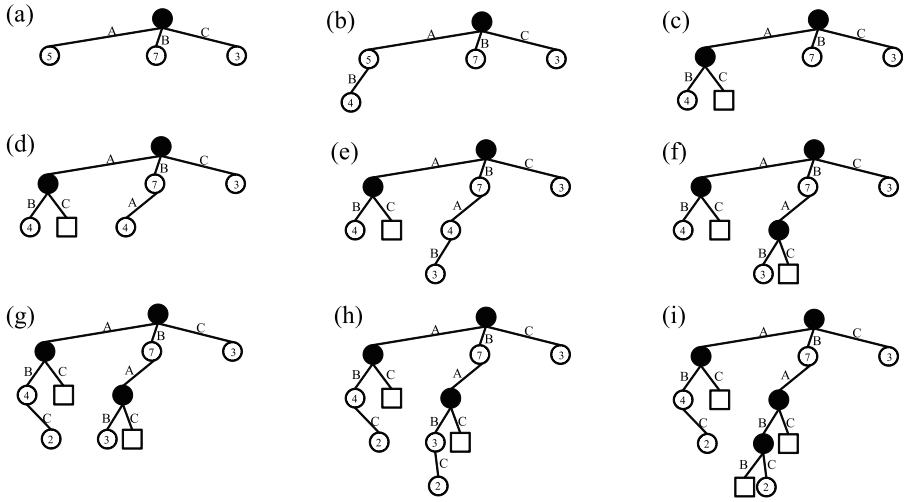


Fig. 10 Proposed algorithm for constructing a parse tree. The black circles represent complete internal nodes, which are not assigned codewords.

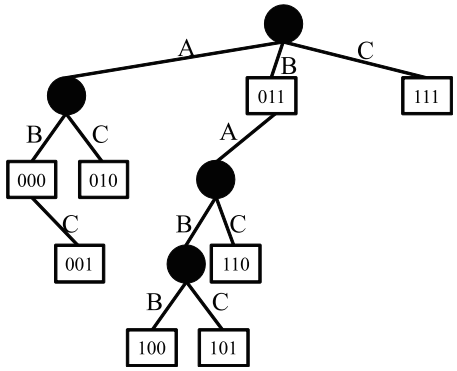


Fig. 9 Parse tree of our method for BABCABABBABCBAC. The squares represent the nodes assigned codewords, corresponding to the numbers in them. The circles represent the complete internal nodes.

quent node from the suffix tree, which has not been included into the parse tree. The algorithm extends the parse tree on a node-by-node basis in contrast to the original STVF algorithm that extends all the children of the chosen node at once. **Figure 9** is an example of the parse tree constructed by the algorithm of Fig. 8 for $T :=$ BABCABABBABCBAC. Now we explain the move of the algorithm. For a given text T, we first construct the suffix tree $ST(T)$ and remove the labels of the leaves in $ST(T)$ except for the first characters of them. Let U be the set of nodes which will be assigned codewords and V be the set of "candidate" nodes for blocks which are in $ST(T)$ but not in the parse tree. Note that each node in V is a child of a node in U. Initially, U is the empty set and V is the children of the root of $ST(T)$. Next, to ensure the algorithm encodes the text correctly, we add all the children of the root to U. Then, we repeat the following procedure while $|U| < 2^k$: we select the node v whose frequency is maximal in V. Then, we add it to U and delete it from V. If there remains just one node $w \in V$ that is a sibling of v, we add w to U and delete its parent from U. It is not necessary to assign a codeword to a complete node because the traversals in the encoding process never fail at any complete nodes. The node p is now complete and thus it will not be assigned a codeword. Finally, we assign unique codewords to the elements in U in a left-to-right manner.

Figure 10 shows the construction process of the parse tree for the running example by the algorithm. The input string is parsed into five substrings by using the parse tree in Fig. 9, as BABC/AB/AB/BABC/BAC, and encoded to 101/000/000/101/110. In this case, the encoded length is shorter than that of the STVF code in the previous section.

We discuss the time and space complexities of the algorithm in Fig. 8. Constructing the suffix tree $ST(T)$ needs $O(|T|)$ time and space. It is obvious that both Line 2 takes $O(1)$ time. Since we can manage both the set U and the tree $\mathcal{T}$ just by marking nodes in $ST(T)$, adding or deleting an element for them is done in $O(1)$ time. To process Line 13 efficiently, we assume that the set V is realized by a priority queue based on a max-heap. That is, we need $O(\log|V|)$ time for adding or deleting an element for V, while answering the maximum element among V is done in $O(1)$ time. Then, Line 3 needs $O(|\Sigma|)$ time. For the loop from Lines 4 to 11, the number of iterations is $O(|\Sigma|)$. Thus, the time complexity of the loop is $O(|\Sigma|\log|\Sigma|)$ since the size of V can increase to $O(|\Sigma|^2)$. For the while loop from Lines 12 to 23, the number of iterations is restricted to the size of U, but the size of V is a dominant factor for the time complexity. We can calculate in $O(1)$ time for each line within the loop except for Lines 16 and 21 [*3]. The number of nodes added to V is $|T|$ at most, and the number of nodes deleted from V too. Thus, Lines 16 and 21 take $O(|T|\log|T|)$ time totally. Finally, Line 24 takes $O(|T|)$ time. Therefore, we can calculate the algorithm in $O(|\Sigma|\log|\Sigma| + |T|\log|T|)$ time totally. The complexity will be $O(|T|\log|T|)$ when $|\Sigma| \le |T|$. For the space consumption, we need only $O(|T|)$ space since both U and $\mathcal{T}$ can be managed by adding $O(1)$ size information on each node of $ST(T)$, in addition to the priority queue whose maximal size is restricted to $|V|$, namely, $O(|T|)$.

Next, we show the encoding and the decoding algorithms. We need to modify the encoding algorithm because codewords are assigned to internal nodes. It is shown in **Fig. 11**. The algorithm traverses the parse tree while it can move by the character read

[*3] For Line 19, we need an auxiliary data structure on each node to do so. For example, it is realized by a doubly linked list between siblings.

© 2012 Information Processing Society of Japan

Table 1 Outline of the text files used for our experiments.

Texts	size (byte)	$\|\Sigma\|$	Content
bible.txt	4047392	63	The King James version of the bible
world192.txt	2473400	94	The CIA world fact book
E.coli	4638690	4	Complete genome of the E.Coli bacterium

Algorithm new-encode($T, \mathcal{T}$):
Input: A text T and a parse tree $\mathcal{T}$.
Output: An encoded sequence of codewords.

```
1: i ← 0;
2: while i < |T|
3:    v ← root;
4:    while str(v) · T[i] is represented by T do
5:       v ← the node that represents str(v) · T[i];
6:       i ← i + 1;
7:    end while
8:    output the codeword assigned to v;
9: end while
```

Fig. 11 Modified encoding algorithm.

from the input text. If the traversal cannot be made, the algorithm suspends to consume the current character and outputs the codeword of the current node, and then resumes the traversal from the root. This encoding process is not instantaneous. Reading-ahead of just one character is needed. Therefore, we call it the *almost instantaneous encoding*. The algorithms of decoding and storing the parse tree are common to the STVF code except for storing incomplete nodes. We add an extra bit indicating whether the node is complete or not for each node. Then the tree structures of a parse tree of M nodes are encoded to $3M$ bits.

The following lemma is important for the correctness of the encoding algorithm using the parse trees constructed by the algorithm in Fig. 8.

Lemma 3. *Let T be a given text and $\mathcal{T}$ be the parse tree of $\mathcal{T}$ constructed by the algorithm in* Fig. 8. *For any suffix s of T, there exist at least one node in $\mathcal{T}$ which represents a nonempty prefix of s, and there exists one node which represents the longest prefix of s in $\mathcal{T}$ and which is also assigned a codeword.*

Proof. The former is clear because all the children of the root are contained in $\mathcal{T}$. We next prove the latter by a reduction to absurdity. Assume that the node v in $\mathcal{T}$ which represents the longest prefix of s is not assigned a codeword. Then, v is a complete internal node because all the leaves and all the incomplete nodes are assigned codewords. However, since all the children of any complete nodes exists in $\mathcal{T}$, it contradicts our assumption that there exists a descendant of v which represents a longer prefix of s than $str(v)$. □

4.3 Experimental Results

We have implemented the Tunstall code, the STVF code, and the almost instantaneous STVF code. We abbreviate them to Tunstall, STVF, and AISTVF respectively. All the programs are written in C++ and compiled by g++ of GNU, version 4.3. The output files of STVF and AISTVF include parse trees. We ran our experiments on an Intel Pentium 4(R) processor of 3.00 GHz and 2 GB of RAM, runnning Debian GNU/Linux 5.0. The texts to be used are selected from "the Canterbury corpus [*4]." For each detail, please refer to **Table 1**.

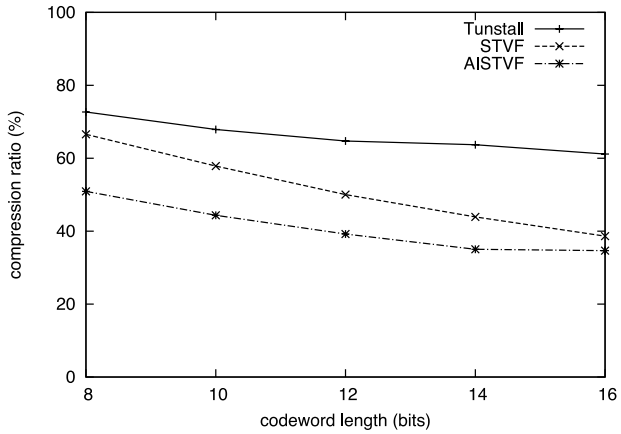


Fig. 12 Compression ratio against codeword length.

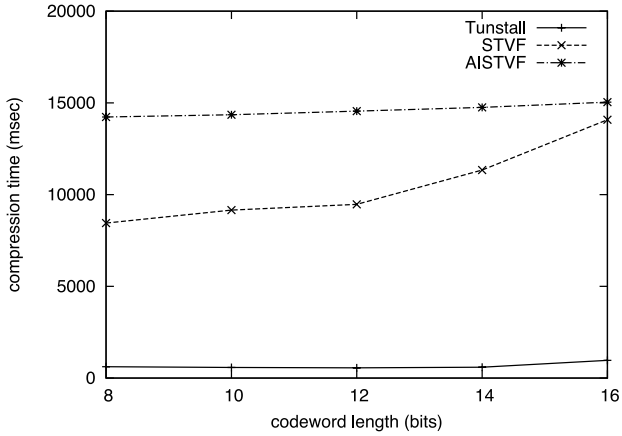


Fig. 13 Compression time against codeword length.

In order to decide the best length of codewords, we first experimented on compression ratios, compression times, and decompression times against the codeword length k. In this experiment, we used only bible.txt. **Figure 12** shows the compression ratios that is defined as (compressed file size)/(original file size). AISTVF achieved a better performance than the others, especially when the codeword length is short. **Figure 13** compares the compression times. We measured the CPU times by the time command of Linux. As shown in this figure, Tunstall is the fastest. The results in **Fig. 14** of decompression times are opposite to the results of compression times. Tunstall required much time for decompression. All the algorithms achieved their best compression ratio when $k = 16$. Thus, we set the codeword length to 16 in the following experiment.

Next, we have compared five compression algorithms: Tunstall, STVF, AISTVF, BPEX, and bzip2. Since bzip2 is widely used and is one of the state-of-the-art compression tools, we include bzip2 as a reference. We used the three texts in Ta-

[*4] http://corpus.canterbury.ac.nz/descriptions/

© 2012 Information Processing Society of Japan

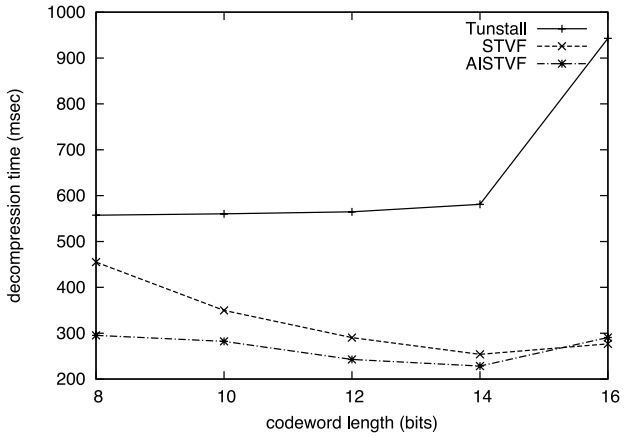


Fig. 14 Decompression time against codeword length.

Table 2 Compression ratios.

method	bible.txt	world192.txt	E.coli
Tunstall	61.16%	69.97%	25.00%
STVF	42.13%	49.93%	28.90%
AISTVF	34.67%	43.97%	28.89%
BPEX	28.05%	26.58%	28.72%
bzip2	20.89%	19.79%	26.97%

Table 3 Compression times (msec).

method	bible.txt	world192.txt	E.coli
Tunstall	965	651	7718
STVF	14185	9398	9028
AISTVF	14942	8196	18994
BPEX	25957	18611	15914
bzip2	1310	851	1505

Table 4 Decompression times (msec).

method	bible.txt	world192.txt	E.coli
Tunstall	944	623	7845
STVF	279	206	268
AISTVF	291	223	320
BPEX	335	209	361
bzip2	537	361	794

ble 1 as test data. The compression ratios are shown in **Table 2**. AISTVF improves the compression ratio approximately by 18% on bible.txt, and by 12% on world192.txt in comparison with STVF. Tunstall compresses most effectively on E.coli. BPEX totally achieves high compression ratios among VF codes. The compression times are shown in **Table 3**. STVF and AISTVF are two times faster than BPEX. However, compared with Tunstall, STVF and AISTVF take much time, because they take much time to construct the suffix tree. The decompression times are shown in **Table 4**. STVF and BPEX take less time in comparison with the others. AISTVF and STVF take less time than the others in almost all the cases.

5. Training Parse Trees

5.1 Reconstruction Algorithm

In this section, we present an algorithm of reconstructing a parse tree to improve the compression ratio. The basic idea is to exchange useless strings in the current parse tree for the other strings not in the parse tree which are expected to be frequently used. Although we must evaluate each string by some measures for doing that, it is quite hard to evaluate precisely in advance as we stated in Section 1. Therefore, we employ a greedy approach;

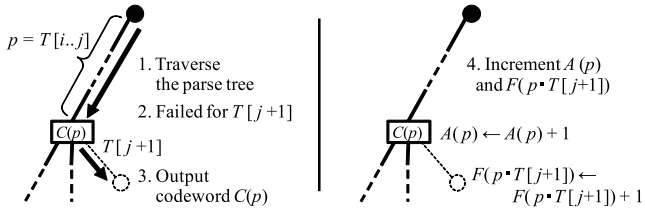


Fig. 15 An example of computing accept counts and failure counts.

Algorithm ReconstructingParseTree(T, D):
Input: A text $T := T[1..n]$ and a set D of strings in the parse tree.
Output: A new set of strings.

```
1: i ← 1, E ← ∅;
2: while i < n
3:    p ← the longest prefix T[i..j] of T[i..n] which is also included
      in D;
4:    A(p) ← A(p) + 1;
5:    if j < |T| then
6:       q ← p · T[j + 1];
7:       E ← E ∪ {q};
8:       F(q) ← F(q) + 1;
9:    end if
10:   i ← j + 1;
11: end while
12: N ← ∅;
13: while D ≠ ∅ and E ≠ ∅
14:    s ← argmin_{s∈D} A(s);
15:    t ← argmax_{t∈E} F(t);
16:    if A(s) < F(t) then
17:       N ← N ∪ {t};
18:       D ← D \ {s};
19:    else
20:       break;
21:    end if
22:    E ← E \ {t};
23: end while
24: return D ∪ N;
```

Fig. 16 Reconstruction algorithm for parse trees.

we reconstruct the parse tree with two empirical measures: the *accept count* and the *failure count*. For any string s in the parse tree, the accept count of s, denoted by $A(s)$, is defined as the number of the occurrences of string s in the encoding. For any string t that is not assigned a codeword, the failure count of t, denoted by $F(t)$, is defined as the number of times that the prefix $t[1..|t|-1]$ of t was in the parse tree and the codeword traversal failed at the last character of t. If $F(t)$ is sufficiently large, it is expected that we can make the average block length longer by including t in the parse tree. The computations of $A(s)$ and $F(t)$ are embeded in the encoding procedure. When $p := T[i..j]$ is parsed in the encoding, $A(p)$ and $F(p \cdot T[j+1])$ are incremented by one simultaneously. **Figure 15** shows an example of computing these measures.

The reconstruction algorithm is shown in **Fig. 16**. Comparing the minimum of $A(s)$ and the maximum of $F(t)$, the reconstruction algorithm repeats exchanging s for t if the former is less than the latter, that is, it removes s from the parse tree and enter t instead. Note that a reconstructed parse tree is not a complete tree any longer, even if the origin is a complete tree like the Tunstall trees. Several internal nodes might be assigned codewords; thus a coding with such a tree becomes an AIVF coding. To train a parse tree we apply the algorithm many times. For each iteration,

© 2012 Information Processing Society of Japan

it first encodes the input data with the current parse tree. Next, it evaluates the contribution of each string in the parse tree, and then exchanges some infrequent strings for the other promising strings.

Next we discuss the time and space complexities of the algorithm in Fig. 16. We assume that the sets D and E are realized by priority queues to calculate Lines 14 and 15 efficiently. For the loop from Line 2 to 11, Line 3 takes $O(|T|)$ time totally, and all Lines except Lines 3 and 7 are done in $O(1)$ time for each. Let E' be the number of parsed blocks, namely, it is equal to $|E|$ after processing the loop. Then, the number of iterations of the loop is $O(E')$, and Line 7 takes $O(E' \log E')$ time totally. For the while loop from Line 13 to 23, Line 18 and Line 22 take $O(\log |D|)$ and $O(\log E')$, respectively. For each iteration, $|E|$ decreases exactly 1 while $|D|$ decreases 1 at most. If $|E| < |D|$ then the number of iterations is just $|E|$, otherwise it is also restricted to $O(|E|)$. Thus, the number of iterations of the while loop is $O(E')$. Therefore, the time complexity of the algorithm is $O(E' \log |D| E' + E' \log E' + |T|)$. Roughly speaking, it is $O(|T| \log |T|)$ since both E' and $|D|$ are $O(|T|)$. For the space consumption, we can prove that it is $O(|T|)$ space from the same discussion on the algorithm in Fig. 8.

5.2 Speeding-up by Sampling

The reconstruction of parse trees discussed above takes much time if the input text is large, since the algorithm scans the whole text many times. If we train with small parts of the text, we can save the training time. Note that we must scan the whole text once to construct the initial parse tree.

We consider training with a string that consists of several *pieces* randomly selected from the text. Using only one part of the input text T, namely a substring of T, does not work well even if we select a substring randomly for each reconstruction, since the parse tree reconstructed by the above algorithm fits too much on the last selection. Using a set of pieces randomly selected from the whole text works well. Let m be the number of pieces, and B be the length of a piece. For given $m \geq 1$ and $B \geq 1$, we generate a sample text S from T at every reconstruction as follows:

$$S := s_1 \cdots s_m \quad (s_k := T[i_k..i_k + B - 1] \text{ for } 1 \leq k \leq m),$$

where i_k is a start position of a piece satisfying $1 \leq i_k \leq |T| - B + 1$. We select the pieces in a uniform random manner for each k. Then, $|S| = mB$ holds. Note that the compression ratios and speeds depend on $|S|$ and m in addition to the number of training iterations.

5.3 Experimental Results

We have implemented the Tunstall codes and the STVF codes with the training approach stated in Section 5, and compared them with BPEX [18], ETDC [5], SCDC [4], gzip, and bzip2. Although ETDC/SCDC are variable-to-variable-length codes, their codewords are byte-oriented and designed for compressed pattern matching. Therefore, we added them in our experiments. We chose 16 as the codeword length of both the STVF codes and the Tunstall codes. Our programs are written in C++ and compiled by g++ of GNU, version 3.4. We ran our experiments on an Intel Xeon (R) 3 GHz and 12 GB of RAM, running Red Hat Enterprise Linux ES Release 4.

Table 5 Outline of the text files used for our experiments.

Texts	size (byte)	$\|\Sigma\|$	Contents
GBHTG119	87,173,787	4	DNA sequences
DBLP2003	90,510,236	97	XML data
Reuters-21578	18,805,335	103	English texts
Mainichi1991	78,911,178	256	Japanese texts (encoded by UTF-16)

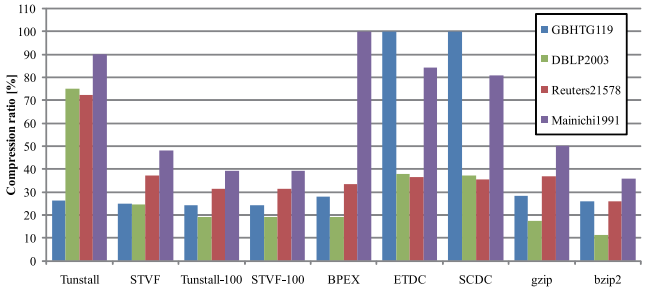


Fig. 17 Compression ratios.

We used DNA data, XML data, English texts, and Japanese texts to be compressed (see **Table 5**). GBHTG119 is a collection of DNA sequences from GenBank [*5], where we eliminated all meta data, spaces, and newline characters. DBLP2003 consists of all the data in the year 2003 from dblp20040213.xml [*6]. Reuters-21578 (distribution 1.0) [*7] is a test collection of English texts. Mainichi1991 [*8] is from Japanese news paper, *Mainichi-Shinbun*, in 1991.

5.4 Compression Ratios and Speeds

The methods in our experiments are the following nine: Tunstall (the Tunstall codes without training), STVF (the STVF codes without training), Tunstall-100 (the Tunstall codes with 100 times training), STVF-100 (the STVF codes with 100 times training), BPEX, ETDC, SCDC, gzip, and bzip2.

Figure 17 shows the results of compression ratios, where every compressed data include the dictionary information. We indicate the compression ratios of the averages of ten executions for Tunstall-100 and STVF-100. STVF, Tunstall-100, and STVF-100 were the best in the compression ratio comparisons for GBHTG119. Since ETDC and SCDC are word based compression methods, they did not work well for the data that are hard to divide into words, such as DNA sequences and Japanese texts. Note that, while Tunstall had no advantage to STVF, Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were better than gzip. **Figure 18** shows the results of compression times. STVF was much slower than Tunstall and ETDC/SCDC since it takes much time for constructing a suffix tree. As Tunstall-100 and STVF-100 took extra time for training, they were the slowest among all for any dataset. **Figure 19** shows the results of decompression times. The decompression times of ETDC, SCDC, and gzip are the shortest, and those of bzip2 and BPEX are the longest. The results of Tunstall and STVF were between those of BPEX and ETDC/SCDC in all the data. The

[*5] http://www.ncbi.nlm.nih.gov/genbank/
[*6] http://www.informatik.uni-trier.de/~ley/db/
[*7] http://www.daviddlewis.com/resources/testcollections/reuters21578/
[*8] http://www.nichigai.co.jp/sales/corpus.html

© 2012 Information Processing Society of Japan

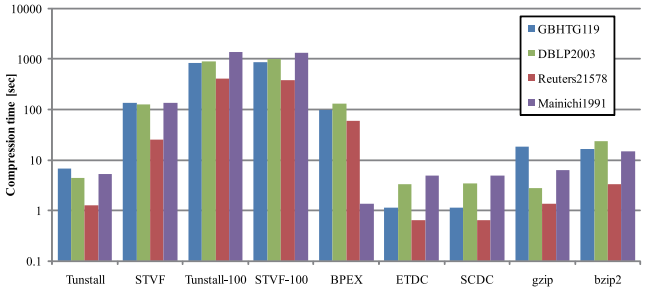


Fig. 18 Compression times.

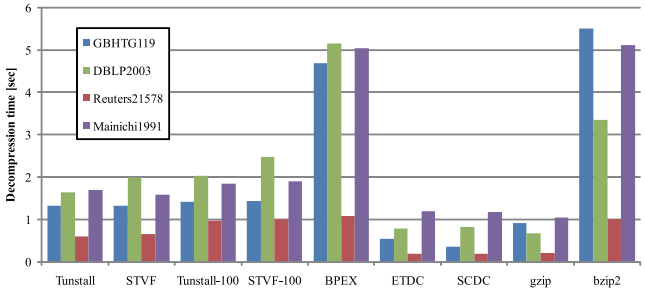


Fig. 19 Decoding times.

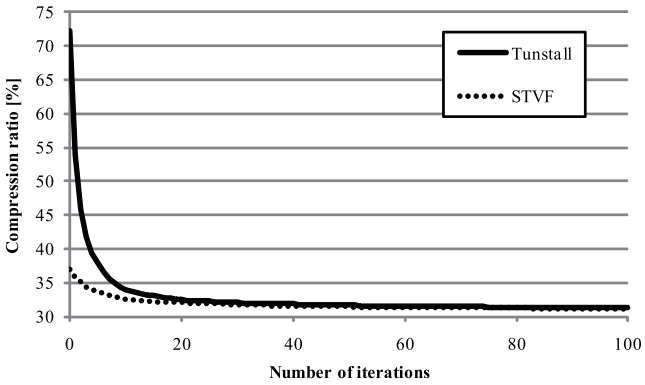


Fig. 20 The effects of training.

decompression procedures of Tunstall-100 and STVF-100 take more time than that of Tunstall and STVF.

5.5 Effects of Training

We examined how many times we should apply the reconstruction algorithm for sufficient training. We chose Reuters-21578 as the test data in the experiments. **Figure 20** shows the result of the effect of training for STVF and Tunstall. The compression ratios of both algorithms were improved rapidly as the number k of reconstruction increases. They seem to come close asymptotically to the same limit, which is about 32%.

We also examined how the sampling technique stated in Section 5.2 effects on compression ratios and speeds. **Figures 21** and **22** show the results for the Tunstall codes with 20 times training. The left side is for compression ratios and the right side is for compression speeds. We measured the average of 100 executions for each result. We observed that the compression ratio achieves almost the same as that of the training method without sampling when the sample size $|S|$ is 25% of the entire text and the number m of pieces is 100. The Tunstall code with training is superior to BPEX in compression ratios when $|S|$ is 20% and $m = 40$. The average compression time of the Tunstall codes at that point was

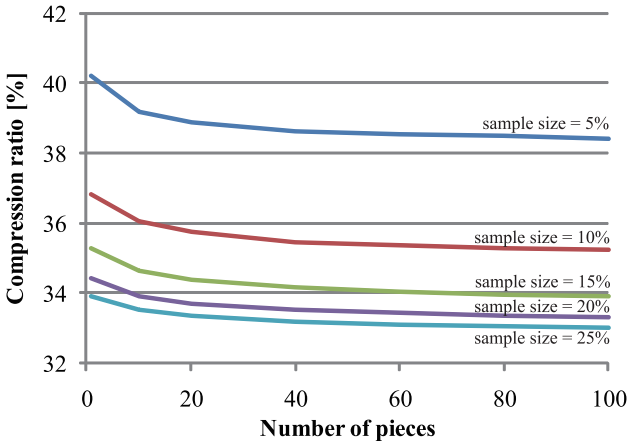


Fig. 21 Training with sampling.

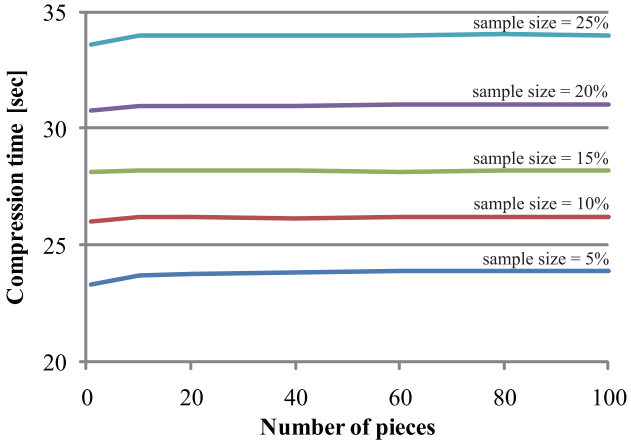


Fig. 22 Training with sampling.

30.97 seconds, while BPEX takes 58.77 seconds.

Although STVF codes are better than the Tunstall codes in compression ratios, it is revealed that the Tunstall codes with training are also useful from the viewpoint of the compression time.

6. Conclusions

We presented several methods of improving VF codes and carried out some experiments for evaluating them: using a frequency-based pruned suffix tree, improving by almost instantaneous coding, and training the parse trees. The experimental results showed that our methods improve in compression ratio in comparison with the traditional VF codes. The compression speed of our methods is faster than that of BPEX and the decompression speed is faster than that of bzip2 and comparative to BPEX in general.

Moreover, we showed experimentally that training the parse trees improves compression ratios of VF codes to the level of state-of-the-art compression methods, such as gzip and BPEX. The Tunstall codes with training are about twice faster than that of BPEX in compression speed when we gain almost the same compression ratios. VF codes with training are stable and are widely applicable to various data: not only English language texts, but also Japanese texts, DNA data, and so on.

Pattern matching algorithms are systematically derived from collage system [13] since a VF coded text is represented as a reg-

© 2012 Information Processing Society of Japan

ular collage system. However, such algorithms are slower than that on gzipped texts [29], [30] in practice. Speeding up the pattern matching on VF codes is our future work.

Reference

[1] Apostolico, A. and Fraenkel, A.: Robust transmission of unbounded strings using Fibonacci representations, *IEEE Trans. Information Theory*, Vol.33, No.2, pp.238–245 (1987).

[2] Brisaboa, N., Fariña, A., Navarro, G. and Paramá, J.: Dynamic lightweight text compression, *ACM Trans. Inf. Syst.*, Vol.28, pp.10:1–10:32 (online), DOI: http://doi.acm.org/10.1145/1777432.1777433 (2010).

[3] Brisaboa, N.R., Fariña, A., López, J.-R., Navarro, G. and Lopez, E.R.: A New Searchable Variable-to-Variable Compressor, *DCC*, pp.199–208 (2010).

[4] Brisaboa, N.R., Fariña, A., Navarro, G. and Esteller, M.F.: (S, C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases, *SPIRE*, pp.122–136 (2003).

[5] Brisaboa, N.R., Iglesias, E.L., Navarro, G. and Paramá, J.R.: An Efficient Compression Code for Text Databases, *ECIR*, pp.468–481 (2003).

[6] Chrobak, M., Kolman, P. and Sgall, J.: The greedy algorithm for the minimum common string partition problem, *ACM Trans. on Algorithms*, Vol.1, No.2, pp.350–366 (2005).

[7] Fraenkel, A.S. and Klein, S.T.: Complexity Aspects of Guessing Prefix Codes, *Algorithmica*, Vol.12, No.4/5, pp.409–419 (1994).

[8] Fraenkel, A.S., Mor, M. and Perl, Y.: Is Text Compression by Prefixes and Suffixes Practical?, *Acta Inf.*, Vol.20, pp.371–389 (1983).

[9] Gage, P.: A new algorithm for data compression, *C Users J.*, Vol.12, pp.23–38 (online) (1994), available from ⟨http://portal.acm.org/citation.cfm?id=177910.177914⟩.

[10] Giegerich, R. and Kurtz, S.: From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction, *Algorithmica*, Vol.19, No.3, pp.331–353 (1997).

[11] Kida, T.: Suffix Tree Based VF-Coding for Compressed Pattern Matching, *Proc. Data Compression Conference 2009* (*DCC2009*), p.449 (2009).

[12] Kida, T.: STVF Code: An Efficient VF Coding using Frequency-base-pruned Suffix Tree, *DBSJ Journal*, Vol.8, No.1, pp.125–130 (2009). (written in Japanese).

[13] Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A. and Arikawa, S.: Collage system: A unifying framework for compressed pattern matching, *Theor. Comput. Sci.*, Vol.298, No.1, pp.253–272 (online), DOI: http://dx.doi.org/10.1016/S0304-3975(02)00426-7 (2003).

[14] Klein, S.T.: Improving Static Compression Schemes by Alphabet Extension, *CPM*, pp.210–221 (2000).

[15] Klein, S.T. and Shapira, D.: Improved Variable-to-Fixed Length Codes, *SPIRE '08: Proc. 15th International Symposium on String Processing and Information Retrieval*, Berlin, Heidelberg, Springer-Verlag, pp.39–50 (online), DOI: http://dx.doi.org/10.1007/978-3-540-89097-3_6 (2009).

[16] Klein, S.T. and Ben-Nissan, M.K.: Using Fibonacci Compression Codes as Alternatives to Dense Codes, *DCC*, pp.472–481 (2008).

[17] Larsson, N.J. and Moffat, A.: Offline Dictionary-Based Compression, *Proc. Data Compression Conference 1999* (*DCC '99*), pp.296–305, IEEE (1999).

[18] Maruyama, S., Tanaka, Y., Sakamoto, H. and Takeda, M.: Context-Sensitive Grammar Transform: Compression and Pattern Matching, *Proc. 15th International Symposium on String Processing and Information Retrieval* (*SPIRE 2008*), LNCS, Vol.5280, pp.27–38 (2008).

[19] Munro, J.I.: Space efficient suffix trees, *J. Algorithms*, Vol.39, No.2, pp.205–222 (online), DOI: http://dx.doi.org/10.1006/jagm.2000.1151 (2001).

[20] Nevill-Manning, C., Witten, I. and Maulsby, D.: Compression By Induction of Hierarchical Grammars, *Proc. Data Compression Conference 1994* (*DCC '94*), pp.244–253, IEEE (1994).

[21] Salomon, D.: *Data Compression: The Complete Reference*, Springer, 4th edition (2006).

[22] Savari, S.A. and Gallager, R.G.: Generalized Tunstall codes for sources with memory, *IEEE Trans. on Information Theory*, Vol.43, No.2, pp.658–668 (1997).

[23] Savari, S.A.: Variable-to-Fixed Length Codes for Predictable Sources, *In Proc. DCC98*, pp.481–490 (1998).

[24] Sayood, K. (Ed.): *Lossless Compression Handbook*, Academic Press (2002).

[25] Tjalkens, T.J. and Willems, F.M.J.: Variable to fixed-length codes for Markov Sources, *IEEE Trans. Inf. Theor.*, Vol.33, pp.246–257 (online), DOI: 10.1109/TIT.1987.1057285 (1987).

[26] Tunstall, B.P.: Synthesis of noiseless compression codes, PhD Thesis, *Georgia Institute of Technology*, Atlanta, GA (1967).

[27] Uemura, T., Kida, T., Yoshida, S., Asai, T. and Okamoto, S.: Training Parse Trees for Efficient VF Coding, *Proc. 17th Symposium on String Processing and Information Retrieval* (*SPIRE2010*), LNCS, Vol.6393, pp.179–184 (2010).

[28] Yamamoto, H. and Yokoo, H.: Average-Sense Optimality and Competitive Optimality for Almost Instantaneous VF Codes, *IEEE Trans. on Information Theory*, Vol.47, No.6, pp.2174–2184 (2001).

[29] Yoshida, S. and Kida, T.: On Performance of Compressed Pattern Matching on VF Codes, Technical Report TCS-TR-A-10-48, Hokkaido University (2010).

[30] Yoshida, S. and Kida, T.: On Performance of Compressed Pattern Matching on VF Codes, *Proc. Data Compression Conference* (IEEE, ed.), p.486 (2011).

[31] Ziv, J. and Lempel, A.: A universal algorithm for sequential data compression, *IEEE Trans. Inf. Theory*, Vol.23, pp.337–343 (1977).

[32] Ziv, J. and Lempel, A.: Compression of individual sequences via variable length coding, *IEEE Trans. Inf. Theory*, Vol.24, pp.530–536 (1978).



Satoshi Yoshida received his B.S. degree in Information Science and M.S. degree in Information Science from Hokkaido University in 2009 and 2011, respectively. He is currently a doctoral student in Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University. His research interests include text algorithms, data structures and data compression.



Takashi Uemura is an engineer at Yahoo! JAPAN. He received his B.S. in Engineering from Hokkaido University in 2006. He received his M.S. and Ph.D. in Computer Science from Hokkaido University in 2008 and 2011, respectively. His research interests include text algorithms, Web mining, and distributed computing. He is a member of IEICE.



Takuya Kida received his B.S. degree in Physics, M.S. and Dr. (Information Science) degrees all from Kyushu University in 1997, 1999, and 2001, respectively. He was a full-time lecturer of Kyushu University Library from October 2001 to March 2004. He is currently an associate professor of Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, since April 2004. His research interests include text algorithms and data structures, information retrieval, and data compression. He is a member of IEICE, JSPS, and DBSJ.

© 2012 Information Processing Society of Japan



Tatsuya Asai received his B.S. degree and M.E. degree from Nagoya University in 1999 and 2001 respectively, and received Dr. (Information Science) degree from Kyushu University in 2004. He was engaged in research on XML database systems with Fujitsu Laboratories Ltd. from 2004. His current research interests include XML stream processing and spatio-temporal data mining. He is a member of IPSJ, JSAI, and ACM.



Seishi Okamoto received his B.S., M.S. and Ph.D. degrees from Kyushu University. He is currently the senior manager of intelligent computing at Fujitsu Laboratories. He is also a visiting professor of Graduate School of Information Science and Technology, The University of Tokyo. His research interests include intelligent computing and its applications.

© 2012 Information Processing Society of Japan