𝕋ℂ𝕊-𝕋ℝ-𝔸-**08-36**

𝕋ℂ𝕊 Technical Report

Suffix Tree Based VF-Coding for Compressed Pattern Matching

by

Takuya Kida

Division of Computer Science

Report Series A

November 18, 2008

Hokkaido University
Graduate School of
Information Science and Technology

Email: kida@ist.hokudai.ac.jp

Phone: +81-011-706-7679
Fax: +81-011-706-7680

Suffix Tree Based VF-Coding for Compressed Pattern Matching

Takuya Kida[*]

November 18, 2008

Abstract

In this paper, we study the coding efficiency of VF codes and the computational complexity of compressed pattern matching on them. VF codes as typified by Tunstall code are paid less attention to so far as compared to FV code such as Huffman code due to its mild compression ratio. From a viewpoint of compressed pattern matching, however, VF codes have some desirable features. We also propose a new VF code called ST-VF code which uses a frequency-base-pruned suffix tree as a parse tree. We also discuss an efficient compressed pattern matching algorithm on ST-VF codes, and show it runs in $O(n+R)$ time for scanning with $O(|D|+m^2)$ time and space for preprocessing, where n is the compressed text length, R is the number of occurrences, $|D|$ is the dictionary size, and m is the pattern length. Some experimental results show that the proposed code achieves the compression ratio of about 41% $\sim$ 50%, which is better than Tunstall code and Huffman code. Moreover a preliminary experiment on speed comparison of compressed pattern matching algorithms on Tunstall code and the proposed code are presented.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_1 \dots p_m$ and a text $T = t_1 \dots t_u$, find all the occurrences of P in T, i.e. return the set $\{|x| \mid T = xPy\}$. Exact string matching has been a basic problem in computer science since its beginnings [CR94].

On the other hand, text compression [BCW90] tries to exploit the redundancies of the text to represent it using less space. There are many different compression schemes, including: Huffman code, Run-length, LZ77, LZ78, and so on [Sal04, Say02], among which the Ziv-Lempel family [ZL77, ZL78] is one of the most popular in practice because of their good compression ratios.

[*]Graduate School of Information Science and Technology, Hokkaido University. Kita 14-jo Nishi 9-chome, 060-0814 Sapporo, Japan. +81-11-706-7679. kida@ist.hokudai.ac.jp.

From theoretical and practical views, it is natural to think of merging search and compression. The *compressed matching problem* was first defined in the work of Amir and Benson [AB92] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \dots z_n$, and a pattern P, the compressed matching problem is the problem of finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches.However, the combined requirements of having a searchable and compressed text are not easy to achieve together, as the only solution before the 90's was to process queries by uncompressing the texts and then searching into them.

From late 90's to the beginning of 2000, several methods that achieve both requirements ware appeared [SMT$^+$00, RTT02]. Surprisingly, they could improve the search speed in almost linear to the compression ratio, i.e., they can perform pattern matching on compressed texts faster than an ordinary search algorithm on uncompressed texts. The key point for the achievement is to select a compression method which is suitable for pattern matching at the sacrifice of compression ratio. Such compression method has the following features from the view of compression fields:

- It is a fixed length code (variable-length-to-fixed-length code). Especially constant number of bytes codeword length is better.
- It has a static and/or compact dictionary.

In practical applications, a compression method that has good compression ratio while preserving the above features, is strongly desired.

A variable-length-to-fixed-length code (*VF code* for short) is a coding scheme that assign fixed length codewords to variable length substrings in the original text. A code that assigns variable length codewords for each symbols (or fixed length strings) such as Huffman code, is called as fixed-length-to-variable-length code (*FV code* for short). VF codes as typified by Tunstall code are paid less attentions although they have preferable engineering aspect that all codewords are the same length.

Tunstall code is known to be an entropy coding, i.e., the average codeword length converges to the entropy rate of the information source, as the same as Huffman code [Sav98], and several improvements have been proposed. However, it seems that there is little to no report showing its efficiency on real text data in actual so far.

In this paper, we discuss and revaluate Tunstall code, in addition to propose a new VF code, *ST-VF code*, that uses a frequency-base-pruned suffix tree as a parse tree. The basic idea of Tunstall code is to use a tree called Tunstall tree in order to parse an input text and then translate to a sequence of codewords. In ST-VF coding, we use a tree which is obtained by pruning a suffix tree by its frequency for the input text instead of Tunstall tree. From the viewpoint of compressed pattern matching, we can assume that the whole text to be searched is given. In such a situation, a suffix tree and its variations have prospects of being good parse trees.

In fact the experimental results shows that ST-VF codes achieves significant improvement on compression ratio to Tunstall codes.

We also discuss and show some preliminary experimental results about compressed pattern mathcing on those codes in order to evaluate the usefulness of ST-VF codes.

2 Related Work

Tunstall code is originally proposed in 1967 by Tunstall in his Ph.D. dissertation [Tun67]. In 1990 Ziv shows that VF codes close to the entropy rate more quickly than FV codes for Markov source [Ziv90]. Tjalkens and Willems also studied VF codes for Markov sources and present a valuable technique to implement VF coder and decoder [TW87]. Savari[SG97] presents the efficiency of VF codes for sources with memory.

Yamamoto and Yokoo present an interesting technique in practice for VF codes [YY01]. In VF codes, each codeword is assigned to each leaf in the parse tree. Their idea is to assign codewords to the internal nodes in the parse tree. Although such codewords do not seem to satisfy the prefix condition in the source alphabet, it is not essential for VF codes because each code has same length and can be decoded correctly and instantaneously.

For compressed pattern matching, several researchers have addressed and achieved good performances [KTS+98, NR99, SMT+00]. Especially compressed pattern matching algorithms on Byte Pair Encoding (BPE for short) and on Stopper Encoding (SE for short) are known to run faster than an ordinary matching algorithm on uncompressed texts. Such compression methods tend to sacrifice the compression ratio, and both of BPE and SE have also not good compression ratios compared with even Huffman code.

Very recently, Maruyama *et al.*[MTST08] present an excellent improvement technique for BPE. The key point of their idea is to extend virtually the dictionary space while it is restricted as 256 in BPE, in addition to capture the Markov property of texts.

3 Preliminaries

3.1 Terminology and Notation

Let Σ be a finite alphabet and let Σ^* be the set of all strings over Σ. We denote the length of string $x \in \Sigma^*$ by $|x|$. The string whose length is 0 is denoted by ε, called *empty string*, that is $|\varepsilon| = 0$. We also define that $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The concatenation

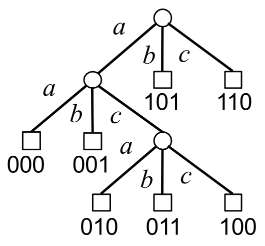

Figure 1: An example of parse tree.

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_1x_2 if no confusion occurs.

Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $w = zyz$, respectively. The ith symbol of a string w is denoted by $w[i]$, and the factor of w that begins at position i and ends at position j is denoted by $w[i : j]$. For convenience, let $w[i : j] = \varepsilon$ for $j < i$. We also denote by $Fac(w)$ the set of all factors of $w \in \Sigma^*$.

3.2 Tunstall Code

Tunstall code [Tun67] uses a parse tree labelled with source symbols instead of a code tree labelled with code symbols like Huffman code.

Let $\Sigma = \{a_1, \ldots, a_k\}$ be a source alphabet of size $k \geq 1$. For an integer $m \geq 1$, we denote by $\mathcal{T}_m$ a trie which is an ordered k-ary tree in which each edge is labelled with each symbol in Σ, and which has m internal nodes. We define the size of the tree by m.

Given a parse tree $\mathcal{T}_m$, all leaves in $\mathcal{T}_m$ are numbered as $\lceil \log N \rceil$ bits integers, where N is the total number of leaves in $\mathcal{T}_m$. The algorithm of source coding with $\mathcal{T}_m$ is as follows:

1. Start the traversal at the root of $\mathcal{T}_m$.

2. Read a symbol one by one from the input text, and traverse the parse tree $\mathcal{T}_m$ by the symbol. If the traversal reaches to a leaf, then output the leaf number.

3. Repeat Step 2 till the text ends.

For example, given a text $T = aaabbacb$ and a parse tree of Fig. 1, the code sequence 000001101011 is obtained. We call a *block* each factor of T parsed by a parse tree. Codeword 011 represents block *acb*, for the running example.

Consider here a memoryless source. Let $Pr(a)$ be an occurrence probability of a source symbol $a \in \Sigma$. Then, a probability of string $x_\mu \in \Sigma^+$ represented by a path from the root to leaf μ is $Pr(x_\mu) = \prod_{\eta \in \xi} Pr(\eta)$, where ξ is the label sequence on the path from the root to μ. From here we identify each node in $\mathcal{T}_m$ and the string represented by the node if no confusion occurs.

In the sense of maximizing the average block length, for a given integer m, an optimal parse tree $\mathcal{T}_m^*$ and its construction are presented by Tunstall. Tree $\mathcal{T}_m^*$ can be constructed recursively as follows:

1. Let $\mathcal{T}_1^*$ be an ordered k-ary tree $\mathcal{T}_1$ whose depth is 1, where $k = |\Sigma|$.
2. For each $i = 2, \ldots, m$, repeat the followings:
   (a) Select leaf $v = v_i^*$ which has the maximal probability $Pr(v)$ among all leaves in $\mathcal{T}_{i-1}^*$.
   (b) Let T_i^* be a tree putting T_1 onto v_i^*.

For any integer $m \geq 1$, we call T_m^* as *Tunstall tree*, and also call VF code that uses T_m^* as *Tunstall code.*

Since the total number N of leaves in T_m^* equals to $m(k-1)+1$, the codeword length ℓ must satisfy $N = m(k-1) \leq 2^\ell$. When we encode texts by codeword of length ℓ, we may construct a parse tree T_m^* of $m = \lfloor (2^\ell - 1)/(k-1) \rfloor$.

Decoding a Tunstall code needs the same parse tree used in its encoding. Compressed texts must include the information of the parse tree. Several succinct representations of trees have been proposed [JSS07, BDM$^+$05, Mun01], and they can code in $2n + o(n)$ bits for n nodes trees. For coding ordered k-ary trees (where k is known), *preorder coding* [KH96] is the simplest and superior to them, because it can code in n bits.

We explain preorder coding below in breif. The coding procedure traverses the target tree from the root by depth-first-search, and outputs 1 if the reached node is internal, otherwise (if it is a leaf) outputs 0 in preorder. For example, the parse tree in Fig. 1 can be coded 1100100000 (information about alphabet Σ is also needed). The total number of output bits equals to the number of all nodes $n = mk+1$ for the tree of size m. Preorder coding is valuable when the tree size is not too large.

Tunstall tree can be reconstructed from information about the tree size m and the frequencies of all symbols in Σ. To store the frequency information in Fk bits will decrease in cost when the parse tree is large enough, where F is the bit length needed for storing a frequency information for each symbol. In ordinal English texts, Fk becomes about 32×100 if we use 32 bits for representing a frequency.

3.3 Suffix Tree

Suffix tree (ST for short) is a data structure that can represent $Fac(T)$ for the string T.It can be denoted by a quadruplet as $ST(T) = (V, root, E, suf)$, where V is a set of nodes, and E is a set of edges and $E \subseteq V^2$. Then, the graph (V, E) forms a rooted tree which have a root node $root \in V$ for $ST(T)$. That is, there exists just one path from $root$ to each node $s \in V$. If $(s,t) \in E$ for nodes $s, t \in V$, we call s the *parent* of

t and call t the *child* of s. We also call the *internal node* of T if it has a child, and call the *leaf* of T if it has no child. Moreover, every node on the path from *root* to the node $s \in V$ is called the *ancestor* of s.

Each edge in E is labelled with a factor of T, represented by a pair of integers (j, k), that is, the *label string* is $T[j : k]$. We denote by $label(s)$ the label string for the edge into $s \in V$. Note that each $label(s)$ is uniquely defined since any child has just one parent. For any $s \in V$, we denote by $\overline{s}$ the string spelled out every ancestor's label from *root* to s, and call it the *string represented by s*. That is, assuming that $root, a_1, a_2, \ldots, s$ is the sequence of the ancestor nodes of s, $\overline{s} = label(root) \cdot label(a_1) \cdot label(a_2) \cdots label(s)$.

For a given text T, each leaf of $ST(T\$)$ corresponds one-to-one with each suffix of T, and any internal node has two children or more, where we assume that \$, called *terminal symbol*, is not included in Σ. Then, there are $|T|$ leaves and less than $2|T|-1$ nodes in $ST(T\$)$. For any two children x, y of each $s \in V$, $\overline{x}$ starts a different character from $\overline{y}$, that is, $\overline{x}[1] \neq \overline{y}[1]$. Then each node $s \in V$ corresponds with a factor of T from the above.

4 Suffix tree based VF code

For a given text T, the deepest leaf of suffix tree $ST(T\$)$ represents T itself. Therefore whole $ST(T\$)$ can not be used as a parse tree. The idea of our new VF code is to prune deep 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 $L \geq k+1$ by pruning. Fig. 2 shows an example of $ST_L(T\$)$. Note that a pruned suffix tree that has all nodes whose depth is 1 in the original $ST(T\$)$, and that it includes any symbols occur in T. Now consider to encode T by codeword length ℓ. As the same as Tunstall code, the formula $L \leq 2^{\ell}$ must be satisfied. The procedure to parse and encode T with tree $ST_L(T\$)$ is also the same as Tunstall code.

The simplest strategy of pruning is to search $ST(T\$)$ by breadth-first-search from the root, and select the most shallow nodes till the number of leaves in a pruned suffix tree is up to L. More sophisticated way is to select nodes so that the occurrence of each string represented by each leaf in the pruned suffix tree becomes uniform. Namely, given the codeword length ℓ and text T, we construct a pruned suffix tree according to the following procedure:

1. Construct 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 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 $T'_i = ST_{L_i}(T\$)$. Here let C_v be the number of direct children of v.

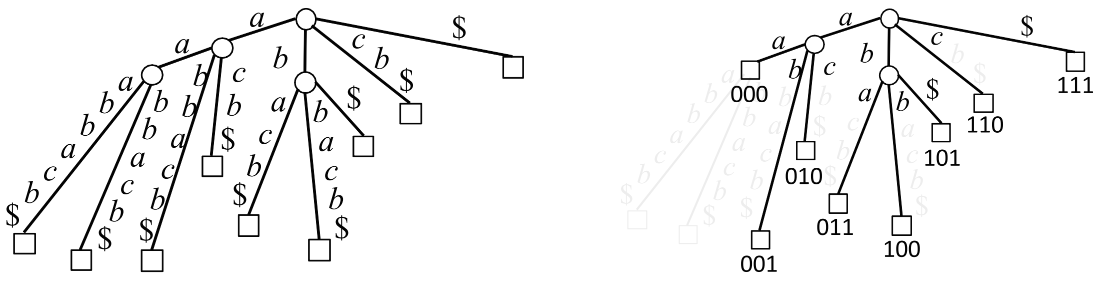

Figure 2: An example of suffix tree and pruned suffix tree.

4. Add all children of v to T'_i as leaves if $L_i + C_v - 1 \leq 2^\ell$, and let it be T'_{i+1} for a new candidate of a parse tree. If child u of v is a leaf in $ST(T\$)$, cut the label on the edge from v to u so that the label length is 1.

5. Repeat Step 2 and 3 till we can not select any node.

Fig. 2 shows a pruned suffix tree, for example, when text $T = aaabbacb\$$ and $L = 8$ are given. The text can be parsed $aa, ab, ba, c, b\$$.

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

Lemma 1 *Using pruned suffix tree $T'_{L'}$ as a parse tree, we can uniquely parse any given text.*

proof. Let D be a set of strings which is entered into pruned suffix tree $T'_{L'}$, and call D as a *dictionary*. Note that from the feature of suffix tree each leaf in $T'_{L'}$ corresponds to each string entered in D. Therefore, to parse text with $T'_{L'}$ means to parse it with the dictionary D.

Consider that prefix $T[1 : i-1]$ of text $T = T[1 : u]$ has already been parsed and coded, and from now we try to parse the rest suffix $T[i : u]$. The next parsed block must be the longest prefix of $T[i : u]$ such that it is also the longest string in the dictionary D. Such a string exists just one in D. If there are more than one, several leaves might represent the same string, and this contradicts the property of suffix tree. Conversely, if there is no such string, it means that no prefix of $T[i : u]$ whose length is greater than or equals to 1 is entered into D. In Step 1 of the above procedure, however, we start with tree $ST_{k+1}(T\$)$ where all nodes in $ST(T\$)$ of depth 1 are included. Therefore, D must include any factor of length 1 at least. Then the proposition consists. $\square$

5 Compressed pattern matching on VF-codes

Consider that text $T = T[1 : u]$ is coded by a VF code into compressed text Z. Then compressed text Z is a sequence of codewords $Z = c_1, c_2, \ldots, c_n$. Note that each codeword $c_i (1 \leq i \leq n)$ corresponds to a factor of T. We denote by $w(c_i)$ the string represented by code c_i.

We define the problem of compressed pattern matching on VF codes as follows:

Definition 1 *Given: a pattern* $P = P[1 : m]$ *and a VF coded text* $Z = c_1, c_2, \ldots, c_n$. *Find: all locations at which* P *occurs in the original text* $T[1 : u] = w(c_1) \cdots w(c_n)$ *without decompressing it.*

When the text is not compressed, we can do pattern matching with an ordinary pattern matching algorithm like KMP method [CR94].

We sketch below the move of our compressed pattern matching algorithm. For a given text T and pattern P, assume that matching of prefix $T[1 : i]$ has been done. Let $1 \leq j \leq m$ be the largest integer such that $T[i - j + 1 : i] = P[1 : j]$. Note that P occurs at the position i if $j = m$. The pattern matching algorithm repeats the followings until T ends: read a next symbol $T[i + 1]$, compute the longest suffix of $T[1 : i + 1]$ that is also a prefix of P, and then report the occurrence if pattern occurs. Let $\delta : \{0, \ldots, p\} \times \Sigma \to \{0, \ldots, p\}$ be the function to compute the length of such unique suffix. The procedure of the pattern matching algorithm is represented as follows:

1. Preprocess for function δ. Set $j := 0$.
2. For each $i = 1, 2, \ldots, u$, repeat the following steps.
   (a) $j := \delta(j, T[i])$.
   (b) Report the occurrence i if $j = m$.

In KMP method, in order to answer $\delta(j, a)$ in constant time, all possible values of $\delta(j, a)$ are precomputed and stored in a table whose size is $O(m)$.

Let $\mathcal{S} = \{0, \ldots, m\}$ be the set of integers, which equals to the range of δ. The domain of $\delta(j, a)$ can be extended from $\delta : \mathcal{S} \times \Sigma$ to $\delta^* : \mathcal{S} \times \Sigma^*$ in natural way as the recurrence formula:

$$\delta^*(j, sa) = \delta(\delta^*(j, s), a), (j \in \mathcal{S}, s \in \Sigma^*, a \in \Sigma).$$

Since a codeword c_i represents a block $w(c_i)$, we can obtain the extension to the domain $\mathcal{S} \times D$ as $\delta'^*(j, c_i) = \delta^*(j, w(c_i))$. The function δ'^* means that it computes at once consecutive δ computations caused by reading $w(c_i)$. However, there are several opportunities that pattern P occurs in $S[i - j + 1 : i] \cdot w(c_i)$. Thus we need another function $o(j, c_i) : \mathcal{S} \times D \to \{1, \ldots, |D|\}$ that returns a set of the occurrences:

$$o(j, c_i) = \left\{ |v| \;\middle|\; \begin{array}{l} v \text{ is a non-empty prefix of } w(c_i) \\ \text{such that } \delta(j, v) = m. \end{array} \right\}.$$

Table 1: About text files to be tested

Texts	size(byte)	$\lvert\Sigma\rvert$	Content
E.coli	4638690	4	Complete genome of the E. Coli bacterium
bible.txt	4047392	63	The King James version of the bible
world192.txt	2473400	94	The CIA world fact book
dazai.utf.txt	7268943	141	The all works of Osamu Dazai (UTF-8 encoded)
1000000.txt	1000000	26	The random string automatically generated

Since the domains of both of δ'^* and o are $\mathcal{S} \times D$, storing all entries of the values for δ'^* and o into a table is not efficient in space when the codeword is quite long. However we can compactly represent the table for δ'^* and o as in [KST$^+$99]. We will omit the detail discussion, VF codes can be framed into collage system proposed by Kida *et al.* [KST$^+$99]. From the above discussion, they are computed in $O(m^2)$ time preprocessing and space to be able to answer in constant time. The following theorem is a modified version of [KST$^+$99].

Theorem 1 (Modified version of [KST$^+$99]) *Let VF be any VF code. Then, the algorithm for compressed pattern matching on VF runs in $O(n + R)$ time after an $O(|D| + m^2)$ time preprocessing using $O(|D| + m^2)$ space, where n is the compressed text length, R is the number of pattern occurrences, $|D|$ is the dictionary size of VF, and m is the pattern length.*

6 Experimental results

We have implemented Tunstall code and ST-VF code. We ran our experiments on an Intel Xeon(R) processor of 3.00GHz dual core and 16GB of RAM running cygwin on Windows Vista.

We have compared three compression algorithms, Huffman code, Tunstall code, and ST-VF code. We have also carried on preliminary experiments about speed comparison of compressed pattern matching algorithms on those three codes. All programs are written in C++ and compiled by g++ of GNU on cygwin, version 3.4.4.

The texts to be used are from two large corpuses, “the Canterbury corpus[1],” and Japanese corpus “J-TEXTS[2],” in addition to a randomly generated text. For each detail, please refer Table 1.

The compression results are shown in Table 2. For ST-VF codes, compressed files include information of a pruned suffix tree. Representing it succinctly contributes to the compression ratio. However we implemented in a simple way in this time, that is, we coded the shape of the tree by balanced parenthesis [Mun01] and spelled

[1]http://corpus.canterbury.ac.nz/descriptions/
[2]http://www.j-texts.com/

Table 2: Experimental results on compression ratios

method	E.coli	bible.txt	world192.txt	dazai.utf.txt	1000000.txt
Original size	4638690	4047392	2473400	7268943	1000000
Huffman	**25.00%**	54.82%	63.03%	57.50%	**59.61%**
Tunstall(8)	27.39%	72.70%	85.95%	100.00%	76.39%
Tunstall(12)	26.47%	64.89%	77.61%	69.47%	68.45%
Tunstall(16)	26.24%	61.55%	70.29%	70.98%	65.25%
ST-VF(8)	25.09%	66.59%	80.76%	73.04%	74.25%
ST-VF(12)	25.10%	50.25%	62.12%	52.99%	68.90%
ST-VF(16)	28.90%	**42.13%**	**49.93%**	**41.37%**	78.99%

Table 3: Experimental results on compression ratios

method	comp. ratio	search time (s)
Huffman	60.52%	2.311
Tunstall(16)	69.34%	2.564
ST-VF(16)	51.28%	2.180

out the sequence of labels with comma-separated. From the table, we can see that ST-VF code is superior than others for larger alphabets, particularly, for English and Japanese texts.

We have also tested compressed pattern matching algorithms on Huffman code, Tunstall code, and ST-VF code. We use "brown.txt from the Brown corpus" whose document size is 6,827,483 bytes and the alphabet size is 96. We used five patterns of length $m = 3$ to 11 which is selected in the text. We estimated the average of CPU times for compressed pattern matching. Table 3 shows the results. In this time, we do not have much time to optimize pattern matching programs on Tunstall code and ST-VF code as they read each code in byte by byte manner. Therefore their I/O times are almost the same. Just compression ratio might contribute their search speed.

7 Conclusion

We have presented a new VF-coding with suffix tree parsing, named ST-VF code. ST-VF code has good compression ratio compared with Huffman code and Tunstall code. It reaches approximately to 41% $\sim$ 50%. On the other hand, we have also presented that ST-VF code and Tunstall code are suitable for compressed pattern matching. Overall, our ST-VF code achieves significant improvement of the compression ratio compared with the previous VF codes such as Tunstall code, while the speed of compressed pattern matching is still as fast as the previous ones.

Since several variations of Tunstall codes are proposed so far, to discuss which codes are more suitable for compressed pattern matching is interesting topics. To compare with other previous works such as BPEX proposed by Maruyama *et al.*[MTST08] is our future work.

Acknowledgements

I thank Prof. Hiroki Arimura for his helpful comments on the first version of this paper. I thank also Prof. Shin-ichi Minato for his advice and encouragement. I wish to acknowledge Mr. Takashi Uemura for offering me a source code of his suffix tree implementation. This work was supported by a Grant-in-Aid for Young Scientists (KAKENHI:20700001) of JSPS.

References

[AB92] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.

[BDM$^+$05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[CR94] M. Croshemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.

[JSS07] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 575–584, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[KH96] K. Kobayashi and T. S. Han. On the pre-order coding for complete k-ary coding trees. In *In Proc. of Inter. Symp. on Information Theory and Its Applications*, pages 302–303, 1996.

[KST$^+$99] Takuya Kida, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symp. on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

[KTS$^+$98] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, pages 103–112, 1998.

[MTST08] Shirou Maruyama, Yohei Tanaka, Hiroshi Sakamoto, and Masayuki Takeda. Context-sensitive grammar transform: Compression and pattern matching. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, Nov. 2008. (to appear).

[Mun01] J. Ian Munro. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[NR99] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. CPM'99*, LNCS 1645, pages 14–36, 1999.

[RTT02] Jussi Rautio, Jani Tanninen, and Jorma Tarhio. String matching with stopper encoding and code splitting. In *In Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 42–52, 2002.

[Sal04] David Salomon. *Data Compression: The Complete Reference.* Springer, 3rd edition, 2004.

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

[Say02] Khalid Sayood, editor. *Lossless Compression Handbook.* Academic Press, 2002.

[SG97] S. A. Savari and R. G. Gallager. Generalized tunstall codes for sources with memory. *IEEE Transactions on Information Theory*, 43(2):658–668, Mar. 1997.

[SMT$^+$00] Y. Shibata, T. Matsumoto, M. Takeda, A. Shiohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *In Proc. 11st Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 181–194, 2000.

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

[TW87] Tjalling J. Tjalkens and Frans M. J. Willems. Variable to fixed-length codes for markov sources. *IEEE Trans. on Information Theory*, IT-33(2), Mar. 1987.

[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.

[Ziv90] Jacob Ziv. Variable-tofixed length codes are better than fixed-to-variable length codes for markov sources. *IEEE Transactions on Information Theory*, 36(4):861–863, July 1990.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.