TCS-TR-A-10-46

TCS Technical Report

Training Parse Trees for Efficient VF Coding

by

Takashi Uemura Satoshi Yoshida Takuya Kida
Tatsuya Asai Seishi Okamoto

Division of Computer Science

Report Series A

August 8, 2010

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

Training Parse Trees for Efficient VF Coding

Takashi Uemura* Satoshi Yoshida* Takuya Kida*
Tatsuya Asai† Seishi Okamoto†

August 8, 2010

Abstract

We address the problem of improving variable-length-to-fixed-length codes (VF codes), which have favourable properties for fast decoding and compressed pattern matching but moderate compression ratios. Their compression ratios depend on the parse trees that they use as a dictionary. However, it is intractable to construct the optimal parse tree, and thus only heuristic approaches can work. We propose a method that trains a parse tree by scanning an input text repeatedly, and we show experimentally that it can improve the compression ratio of VF codes rapidly to the level of state-of-the-art compression methods.

1 Introduction

From the viewpoint of speeding up pattern matching on compressed texts, *variable-length-to-fixed-length codes* (VF codes for short) are reevaluated recently [8, 11]. 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 codeword enables us to touch any parsed block randomly without concerning about codeword boundaries.

Several promising VF codes have been proposed so far. Maruyama *et al.* [13] proposed an excellent compression method, which is a variation of grammar-based compressions. They propose a 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 represeted 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. Klein and

*Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo 060-0814, Japan
†Fujitsu Laboratories Ltd., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan
[1]This name comes from the program implemented by Maruyama.

Shapira [11] and Kida [8] proposed independently VF codes based on suffix tree [7] (STVF code for short). 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 parse tree, and then the text is scanned again and translated into a sequence of codewords. The compression speed of [8] is faster than that of BPEX, and the compression ratio is better than classical VF codes like Tunstall code [17], but not better than BPEX. A VF code that achieves fast compression/decompression and high compression ratio is desired.

Let Σ be an alphabet and k be the codeword bit length. Consider a text $T = t_1 t_2 \cdots t_n$ to be encoded by k-bit fixed length code, 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 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 [11]:

> 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 [6], and other similar problems of devising a code have also been shown to be NP-complete in [4, 5, 10]. A natural approach is thus to suggest heuristical solutions and compare their efficiencies.

Our concern for this problem is how to construct parse trees that approximate the optimal tree better. In most VF codes, a frequency of each substring of T is often used as a clue for the approximation, since it could be related to the number of occurrences in a sequence of parsed blocks. This gives a chicken and egg problem as Klein and Shapira also stated in [11]; that is, to construct a better dictionary, which decides the partition of T, one has to estimate the number of entries that occurs in the partition.

In this paper we discuss about a method for training a parse tree of a VF code to improve its compression ratio. We propose an algorithm of reconstructing a parse tree based on the merit of each node, and we employ a heuristic approach; we apply the reconstruction many times, scanning the input text repeatedly. We can control the number of the scanning time, and also we can employ a random sampling technique to reduce the training time. We show experimentally that our method can improve VF codes comparable to gzip and BPEX with a moderate sacrifice of compression time.

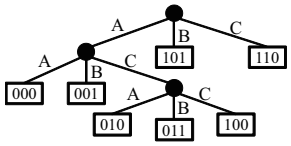

Figure 1: An example of a parse tree.
The squares represent leaves, where codewords are assigned. The circles represent internal nodes and the numbers in the circles are their frequencies.

The rest of this paper is organized as follows. In Section 2, we discuss about VF codes, which includes brief sketches of Tunstall codes. In Section 3, we discuss about STVF codes. In Section 4, we introduce our method of training a parse tree. In Section 5, we show some experimental results and describe our observations about them. Finally, we conclude in Section 6.

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. There are many variations on how they parse the input, what kind of data structures they use as 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.

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 each leaf in $\mathcal{T}$ is numbered as a k-bits integer, where $\ell \leq 2^k$. Then, we can parse and encode T with $\mathcal{T}$ as follows:

1. Start the 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 becomes 000/001/101/011. We call a *block* each factor of T parsed by a parse tree. Codeword 011, for the running example, represents block ACB.

A decoding process of a VF code is quite simple. We can decode by replacing a codeword to a corresponding string as referring the restored parse tree.

For a memory-less information source, *Tunstall code* [17] is known to be an optimal VF code (see also [15]); its average code length par symbol comes asymptotically close to the entropy of the input source when the codeword length goes to infinity. It uses a

parse tree called Tunstall tree, which is the optimal tree in the sense of maximizing the average block length. Tunstall tree is an ordered complete k-ary tree that each edge is labelled with a different symbol in Σ, where $k = |\Sigma|$. 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 $\Pr(x_\mu) = \prod_{\eta \in \xi} \Pr(\eta)$, where ξ is the label sequence on the path from the root to μ (from now on we identify a node in $\mathcal{T}$ and a string represented by the node if no confusion occurs). Then, Tunstall tree $\mathcal{T}^*$ can be constructed as follows:

1. Initialize $\mathcal{T}^*$ as the ordered k-ary tree whose depth is 1, which consists of $k+1$ nodes, where $k = |\Sigma|$.
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 a maximum probability among all leaves in $\mathcal{T}^*$.
   (b) Make v be an internal node by adding k children onto v.

Let m be the number of internal nodes in $\mathcal{T}^*$. Since the number of leaves in $\mathcal{T}^*$ equals to $m(k-1)+1$, which is less than or equal to 2^ℓ. Hence, $m = \lfloor (2^\ell - 1)/(k-1) \rfloor$. For the other information sources, like a source with memory [14, 16], there have been proposed several coding methods that are based on Tunstall code.

Although the preprocessing time for pattern matching on a VF code depends on the size of the parse tree and the data structures for storing it, we can consider that the matching speed is almost in proportion to the compression ratio. The reason is that the time for scanning an input encoded text dominates the total time for pattern matching when the input is enough large. Therefore, the pattern matching becomes faster as the compressed data size becomes smaller; a higher compression ratio leads a smaller amount of data to be processed. Of course, the matching speed depends on what sort of algorithm we use. From the theoretical viewpoints, the VF codes we discussed above can be classified as a regular collage system [9]; thus we can obtain systematically an algorithm of Aho-Corasick type or Boyer-Moore type.

3 STVF codes

A Suffix Tree based VF code (STVF code for short[2]) is a coding that constructs a suitable parse tree for the input text by using a suffix tree, which is a well-known index structure that stores all substrings in the target text compactly. It is, namely, an off-line compression scheme that encodes after gathering the statistical information of the whole input text beforehand. Since the suffix tree for the input text includes

[2]Strictly, the methods of [8] and [11] are slightly different in detail. However, we call them the same name here since the key idea is the same.

the text itself, we can not use the whole tree as a parse tree. We must prune it with some frequency-base heuristics to make a compact and efficient parse tree.

In the original STVF coding, codewords are assigned only to leaves in a parse tree. Some codewords are assigned to short and infrequent substrings, which cause a decline in the compression ratio. If we can assign codewords to the internal nodes, we can prune such useless leaves from the parse tree. To do this we modify the encoding procedure as follows:

1. The procedure traverses the parse tree while it can move by a symbol read from the input text.
2. If the traversal fails, then the procedure outputs the codeword of the current node without consuming the current symbol,
3. and then resumes the traversal from the root.

This encoding process is not instantaneous. Reading-ahead of just one symbol is needed. This type of VF coding is called *almost-instantaneous VF coding* (AIVF coding for short).

An AIVF coding enables us to remove infrequent edges, namely infrequent substrings, from the parse tree, and to leave only frequent edges. This flexible selection of dictionary entries contributes to an improvement in the compression ratio. We have proposed a coding method that we bring the idea of AIVF coding into STVF coding [19]. We call this variation as a STVF code hereafter instead of the original one[3]. We will explain the algorithm of constructing a parse tree of STVF code below.

First of all, we will make a brief sketch of suffix tree, which is the basis of the parse tree for STVF coding. For a given text T, the *suffix tree* $ST(T)$ is a compacted trie that represents all the suffixes of T. Note that $ST(T)$ can be constructed in $O(|T|)$ time and space [20]. 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 labelled by a non-empty substring of T.
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. Then, any substring of T is a prefix of the representing string of a node in $ST(T)$.

[3]This variation also employs a dynamic pruning technique stated in [18] to improve the compression speed and memory usage with a little sacrifice of the compression ratio.

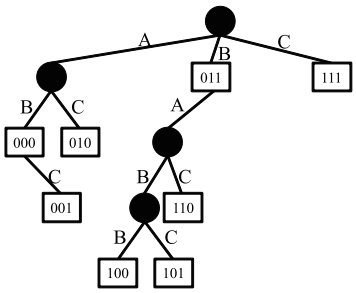

Figure 2: Parse tree of (improved) STVF coding for $T = \mathrm{BABCABABBABCBAC}$.
The squares represent the nodes assigned codewords, corresponding to the numbers in them. The circles represent the complete internal nodes.

For a node v in $ST(T)$, the *frequency* of v is defined as the number of occurrences of $str(v)$ in T, and denoted by $f(v)$. Since $f(v)$ can be obtained as the number of leaves in the subtree rooted at v, we can compute all of them in $O(|T|)$ time by a post-order traversal at once.

Next, we outline the algorithm of constructing a parse tree for a STVF code. The idea is to repeat choosing a node whose frequency is the highest in the suffix tree but not yet in the parse tree. The construction algorithm extends the parse tree on a node-by-node basis. We say that an internal node u in the parse tree is *complete* if the parse tree contains all the children of u in $ST(T)$. We do not need to assign a codeword to any complete node, since the encoding process never fail its traversals at a complete node. Figure 2 is an example of the parse tree constructed by the algorithm of [19] for $T = \mathrm{BABCABABBABCBAC}$. We can parse T to five substrings with the parse tree in Fig. 2, as BABC/AB/AB/BABC/BAC, which are encoded to 101/000/000/101/110.

For Tunstall codes and STVF codes, as we need a parse tree when we decompress an encoded text, we have to store the information for it in addition to a sequence of codewords. For the former, all that we have to store is only the frequencies of all symbols in the alphabet, since we assume that the model of the text is a memory-less source; we can reconstruct the same tree from the frequencies. For the latter, we have to store the whole parse tree that is constructed at the encoding process. The size of the tree increases exponentially with the length of codewords. Therefore, we need to decide a suitable codeword length for compressing a text well. A practical range is about from 7 to 18 for natural language texts, DNA data, and so on. From the viewpoint of compressed pattern matching, the lengths of 8 or 16 would be the best, since we do not need any recognition of codeword boundaries and moreover we can treat the encoded text in a byte-by-byte manner.

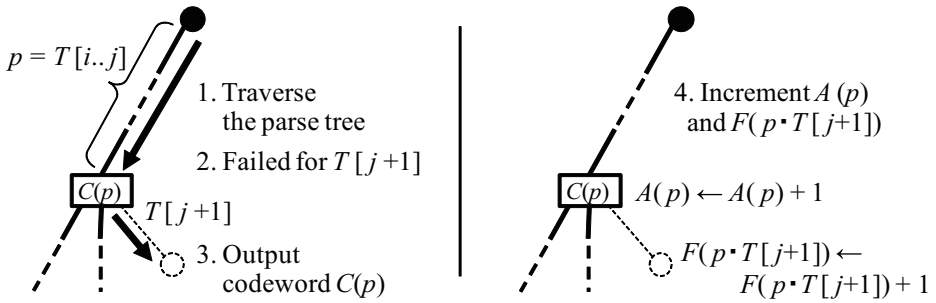

Figure 3: An example of computing accept counts and failure counts.

4 Training parse trees

4.1 Reconstruction algorithm

In this section, we present a reconstruction algorithm for a readymade parse tree to improve its compression ratio. The basic idea is to exchange useless strings in the current parse tree as a result for the other strings that 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 Sec. 1. Therefore, we employ a greedy approach; we reconstruct the parse tree with two empirical measures.

We define two measures for evaluating strings. For any string s in the parse tree, the *accept count* of s, denoted by $A(s)$, is defined as the number of that s was used 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 that the prefix $t[1..|t|-1]$ of t was used but the codeword traversal failed at the last character of t. That is, $F(t)$ suggests how often t likely be used if t is in the parse tree. We can embed the computations of $A(s)$ and $F(t)$ 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. Figure 3 shows an example of computing these measures.

Comparing the minimum of $A(s)$ and the maximum of $F(t)$, the reconstruction algorithm repeats to exchange s and t if the former is less than the latter; it removes s from the parse tree and enter t instead. The algorithm is as in Fig.4. Note that a reconstructed parse tree is not a complete tree any longer, even if its origin is complete like 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, it first encodes the input data with 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.

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;
```

Figure 4: Reconstruction algorithm for parse trees.

4.2 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 can train with small parts of the text, we can save the training time. Note here that we have to scan the whole text once to construct the initail parse tree.

Let T be the input text. We consider to train with a string that consists of several *pieces* randomly selected from the text. Using only a part of T, namely a substring of T, does not work well even if we select randomly for each iteration, 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 can work 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 iteration 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 $1 \leq i_k \leq |T| - B + 1$ is a start position of a piece that we select in a uniform

Table 1: About text files to be used.

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)

random manner for each k. Then, $|S| = mB$. Note that the compression ratios and speeds depend on $|S|$ and m in addition to the number of training iterations.

5 Experimental Results

We have implemented Tunstall coding and STVF coding with training approach that we stated in Sec. 4, and compared them with BPEX [13], ETDC [3], SCDC [2], gzip, and bzip2. Although ETDC/SCDC are variable-to-variable length codes, their codewords are byte-oriented and designed for compressed pattern matching. We chose 16 as the codeword lengths of both STVF coding and Tunstall coding. 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.

We used DNA data, XML data, English texts, and Japanese texts to be compressed (see Table 1). GBHTG119 is a collection of DNA sequences from GenBank[4], which is eliminated all meta data, spaces, and line feeds. DBLP2003 consists of all the data in 2003 from dblp20040213.xml[5]. Reuters-21578(distribution 1.0)[6] is a test collection of English texts. Mainichi1991[7] is from Japanese news paper, *Mainichi-Shinbun*, in 1991.

5.1 Compression ratios and speeds

The methods we tested are the following nine: Tunstall (Tunstall codes without training), STVF (STVF codes without training), Tunstall-100 (Tunstall codes with 100 times training), STVF-100 (STVF codes with 100 times training), BPEX, ETDC, SCDC, gzip, and bzip2. Figure 5 shows the results of compression ratios, where every compression ratio includes dictionary informations. We measured the averages of ten executions for Tunstall-100 and STVF-100.

For GBHTG119, STVF, Tunstall-100, and STVF-100 were the best in the compression ratio comparisons. Since ETDC and SCDC are word-base compression, they

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

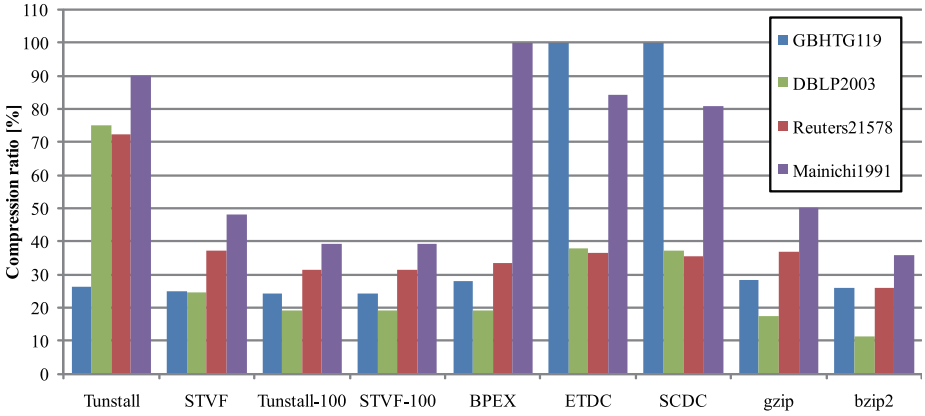

Figure 5: Compression ratios.

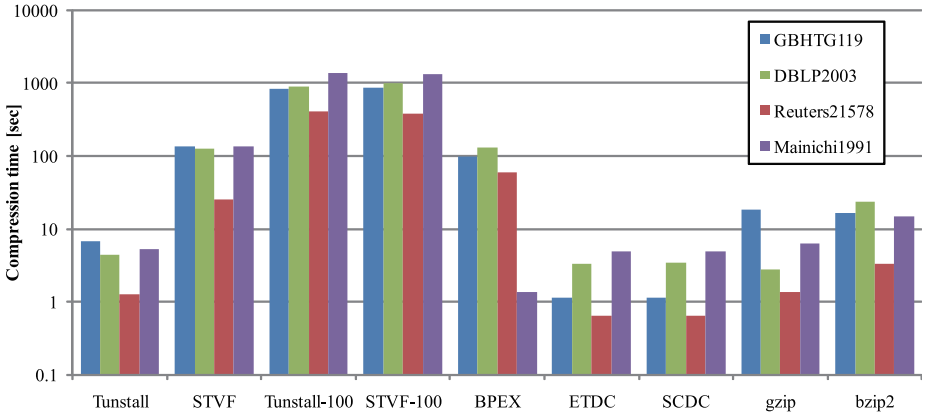

Figure 6: Compression times.

could not work well for the data that are hard to parse, such as DNA sequences and Unicode texts. Note that, while Tunstall had no advantage to STVF, Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were between gzip and bzip2.

Figure 6 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 7 shows the results of decompression times. Tunstall and STVF were between BPEX and ETDC/SCDC in all the data. Tunstall-100 and STVF-100 became slightly slow.

5.2 Effects of training

We examined how many times we should apply the reconstruction algorithm for sufficient training. We chose Reuter21578 as the test data in the experiments. Figure 8

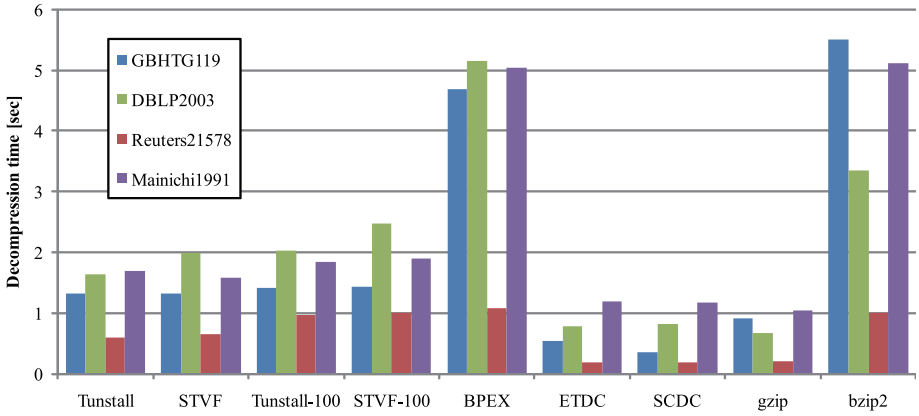

Figure 7: Decoding times.

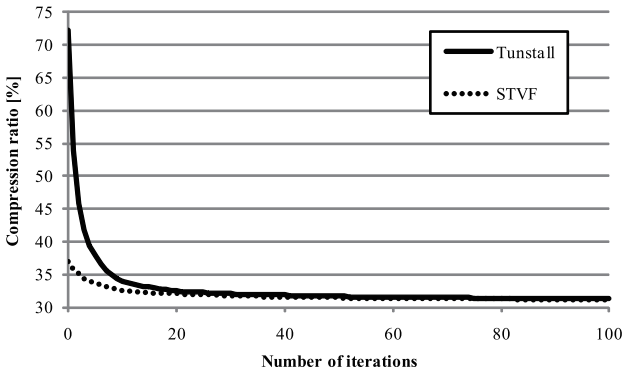

Figure 8: The effects of training.

shows the results of the effect of training for STVF and Tunstall. We can see that both compression ratios were improved rapidly as the number k of iterations increases. We can also see that they seem to come close asymptotically to the same limit, which is about 32%.

We also examined how the sampling technique stated in Sec. 4.2 effects on compression ratios and speeds. Figure 9 shows the results for 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 compressino ratio can achieve almost the same limit when the sampling size $|S|$ is 25% of the text and the number m of pieces is 100. Compared with BPEX, Tunstall codes with training can overcome in compression ratios when $|S|$ is 20% and $m = 40$. The average compression time of them at that point was 30.97 seconds, while that of BPEX was 58.77 seconds.

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

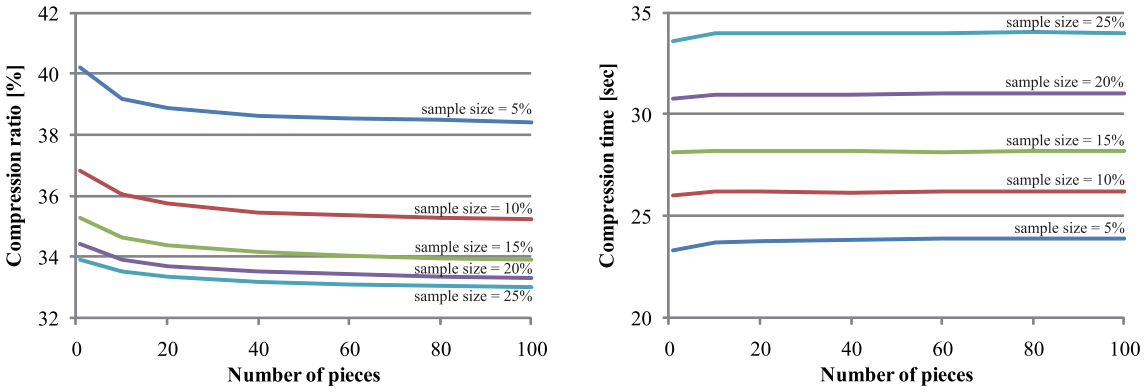

Figure 9: Training with sampling.

6 Conclusions

We present a heuristic method of training a parse tree of a VF code to improve its compression ratio. We showed experimentally that our method can improve compression ratios of VF codes to the level of state-of-the-art compression methods, such as gzip and BPEX. 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 wide applicable to various data: not only English language texts, but also Unicode texts, DNA data, and so on. To compare with the variable-to-variable codes like [1] and [12], which are also designed for compressed pattern matching, is our future work.

References

[1] Brisaboa, N.R., Fariña, A., López, J.R., Navarro, G., Lopez, E.R.: A new searchable variable-to-variable compressor. In: DCC. pp. 199–208 (2010)

[2] Brisaboa, N.R., Fariña, A., Navarro, G., Esteller, M.F.: (s, c)-dense coding: An optimized compression code for natural language text databases. In: SPIRE. pp. 122–136 (2003)

[3] Brisaboa, N.R., Iglesias, E.L., Navarro, G., Paramá, J.R.: An efficient compression code for text databases. In: ECIR. pp. 468–481 (2003)

[4] Chrobak, M., Kolman, P., Sgall, J.: The greedy algorithm for the minimum common string partition problem. ACM Transactions on Algorithms 1(2), 350–366 (2005)

[5] Fraenkel, A.S., Klein, S.T.: Complexity aspects of guessing prefix codes. Algorithmica 12(4/5), 409–419 (1994)

[6] Fraenkel, A.S., Mor, M., Perl, Y.: Is text compression by prefixes and suffixes practical? Acta Inf. 20, 371–389 (1983)

[7] Giegerich, R., Kurtz, S.: From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. Algorithmica 19(3), 331–353 (November 1997)

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

[9] Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: a unifying framework for compressed pattern matching. Theor. Comput. Sci. 298(1), 253–272 (2003)

[10] Klein, S.T.: Improving static compression schemes by alphabet extension. In: CPM. pp. 210–221 (2000)

[11] Klein, S.T., Shapira, D.: Improved variable-to-fixed length codes. In: SPIRE '08: Proceedings of the 15th International Symposium on String Processing and Information Retrieval. pp. 39–50. Springer-Verlag, Berlin, Heidelberg (2009)

[12] Klein, S.T., Ben-Nissan, M.K.: Using fibonacci compression codes as alternatives to dense codes. In: DCC. pp. 472–481 (2008)

[13] Maruyama, S., Tanaka, Y., Sakamoto, H., Takeda, M.: Context-sensitive grammar transform: Compression and pattern matching. In: Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008). LNCS, vol. 5280, pp. 27–38 (Nov 2008)

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

[15] Savari, S.A.: Variable-to-fixed length codes for predictable sources. In: In Proc. of DCC98. pp. 481–490 (1998)

[16] Tjalkens, T.J., Willems, F.M.J.: Variable to fixed-length codes for markov sources. IEEE Trans. on Information Theory IT-33(2) (Mar 1987)

[17] Tunstall, B.P.: Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Inst. Technol., Atlanta, GA (1967)

[18] Uemura, T., Kida, T., Arimura, H.: An approximate substring index for text stream. In: DEIM Forum 2009. pp. E8–6 (2009), (written in Japanese)

[19] Uemura, T., Yoshida, S., Kida, T.: An improvement of stvf code by almost instantaneous encoding. Tech. rep., Hokkaido University, Division of Computer Science (2010)

[20] Weiner, P.: Linear pattern matching algorithms. In: In Proc. of the 14th IEEE Symp. on Switching and Automata Theory. pp. 1–11 (1973)