Training Parse Trees for Efficient VF Coding

Takashi Uemura[1], Satoshi Yoshida[1], Takuya Kida[1],
Tatsuya Asai[2], and Seishi Okamoto[2]

[1] Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo 060-0814, Japan
[2] Fujitsu Laboratories Ltd., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan

Abstract. We address the problem of improving variable-length-to-fixed-length codes (VF codes), which have favourable properties for fast compressed pattern matching but moderate compression ratios. Compression ratio of VF codes depends on the parse tree that is used as a dictionary. We propose a method that trains a parse tree by scanning an input text repeatedly, and we show experimentally that it improves 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) have been reevaluated lately [3, 4]. 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. It is quite hard to construct the optimal parse tree that gives the best compression ratio for the input text, since it is equal to or more difficult than NP-complete [4].

Our concern 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 stated in [4]; 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. We employ a heuristic approach: scanning the input text for estimating the parse tree and then reconstructing it many times. We can control the number of repetition and also we can employ a random sampling technique to reduce the training time. We show experimentally that our method improves VF codes comparable to gzip and the others with a moderate sacrifice of compression time.

2 Variable-length-to-Fixed-length codes

Let Σ be a finite alphabet. A VF code is a source coding that parses an input text $T \in \Sigma^*$ into a consecutive sequence of variable-length substrings and then assigns a fixed length codeword to each substring. We will describe the brief sketches of two VF codes below.

Tunstall code [7] is an optimal VF code (see also [6]) for a memory-less information source. 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 the root and its children; it has $k+1$ nodes.
2. Repeat the following while the number of leaves in $\mathcal{T}^*$ is less than 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.

A Suffix Tree based VF code [3, 4] (STVF code for short[3]) is a coding that constructs a suitable parse tree for the input text by using a suffix tree [8], which is a well-known index structure that stores all substrings and their frequencies in the target text compactly. In STVF codes, a suffix tree for the input text is constructed at first, and then the frequencies of all nodes are precomputed. Since the suffix tree for the input text includes the text itself, the whole tree can not be used as a parse tree. We have to prune it with some frequency-based heuristics to make a compact and efficient parse tree.

We outline the algorithm of constructing the parse tree. The algorithm starts with the initial parse tree that contains the root and its k children in the suffix tree. Then, it repeats choosing a node whose frequency is the highest in the suffix tree but not yet in the parse tree, and putting it into the parse tree. The construction algorithm extends the parse tree on a node-by-node basis.

An internal node u in the parse tree is said to be *complete* if the parse tree contains all the children of u in the suffix tree. We do not need to assign a codeword to any complete node, since the encoding process never fail its traversals at a complete node. In Tunstall codes and the original STVF codes, all the internal nodes are complete; only leaves are assigned codewords. An idea of improving VF codes is to include incomplete nodes in the parse tree, but we have to modify the coding process so that it works in a non-instantaneous way. We omit the detail of the modified coding process for lack of space.

[3] Strictly, the methods of [3] and [4] are slightly different in detail. However, we call them the same name here since the key idea is the same.

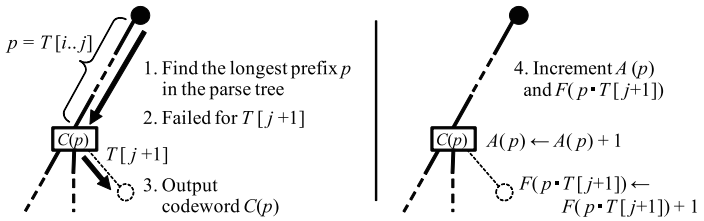

Fig. 1. An example of computing accept counts and failure counts.

3 Training parse trees

In this section, we present a reconstruction algorithm for a ready-made parse tree to improve the 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.

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 1 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.

To train a parse tree we apply the algorithm many times. For each iteration, it first encodes the input text 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.

4 Experimental Results

We have implemented Tunstall coding and STVF coding with training approach that we stated in Sec. 3, and compared them with BPEX [5][4], ETDC [2], SCDC [1], 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.

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

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)

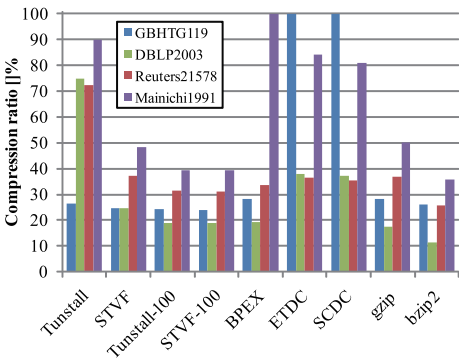

Fig. 2. Compression ratios.

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[5], which is eliminated all meta data, spaces, and line feeds. DBLP2003 consists of all the data in 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.

4.1 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 2 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-based compression, they 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 advan-

[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`

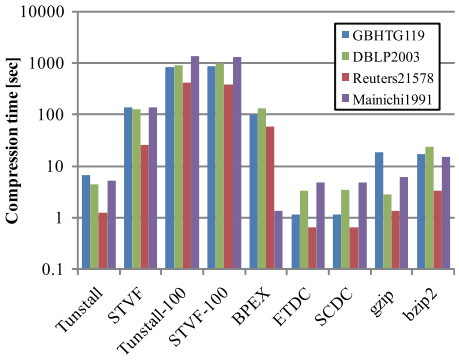

Fig. 3. Compression times. Note that the vertical axis is logarithmic scale.

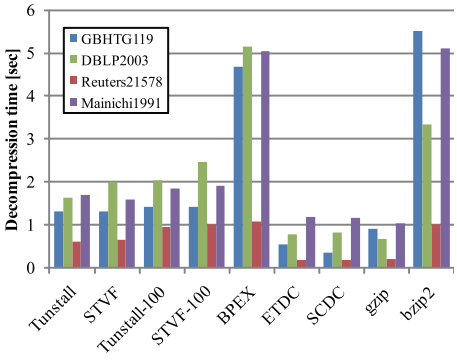

Fig. 4. Decoding times.

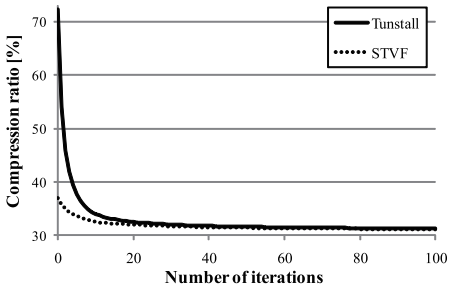

Fig. 5. The effects of training.

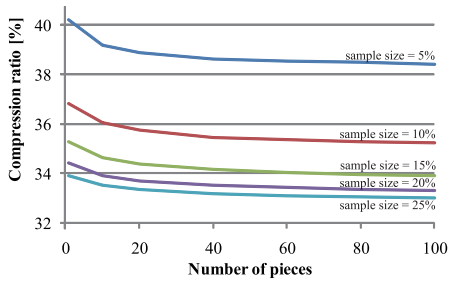

Fig. 6. Training with sampling.

tage to STVF, Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were between gzip and bzip2.

Figure 3 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 4 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.

4.2 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 5 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%.

4.3 Speeding-up by sampling

In this experiment, we introduce a random sampling technique to save the training time.

Let T be the input text, 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 $S = s_1 \cdots s_m$ ($s_k = T[i_k..i_k + B - 1]$ for $1 \leq k \leq m$), where $1 \leq i_k \leq |T| - B + 1$ is a start position of a piece s_k that we select in a uniform random manner for each k. Then, $|S| = mB$.

Figure 6 shows the compression ratios for Tunstall codes with 20 times training. We measured the average of 100 executions for each result. We observed that the compression ratio achieves 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 overcome in compression ratios when $|S|$ is 20% and $m = 40$. The average compression time 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 viewpoint of compression time.

Acknowledgements

This work was partly supported by a Grant-in-Aid for JSPS Fellows (KAKENHI:21002025) and a Grant-in-Aid for Young Scientists (KAKENHI:20700001) of JSPS.

References

1. 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 2003. LNCS, vol. 2857, pp. 122–136. Springer, Heidelberg (2003)
2. Brisaboa, N.R., Iglesias, E.L., Navarro, G., Paramá, J.R.: An efficient compression code for text databases. In: ECIR 2003. LNCS, vol. 2633, pp. 468–481. Springer, Heidelberg (2003)
3. Kida, T.: Suffix tree based VF-coding for compressed pattern matching. In: Data Compression Conference 2009. p. 449. IEEE Computer Society, Los Alamitos, CA (Mar 2009)
4. Klein, S.T., Shapira, D.: Improved variable-to-fixed length codes. In: SPIRE 2008. LNCS, vol. 5280, pp. 39–50. Springer, Heidelberg (2008)
5. Maruyama, S., Tanaka, Y., Sakamoto, H., Takeda, M.: Context-sensitive grammar transform: Compression and pattern matching. In: SPIRE 2008. LNCS, vol. 5280, pp. 27–38. Springer, Heidelberg (2008)
6. Savari, S.A.: Variable-to-fixed length codes for predictable sources. In: Data Compression Conference 1998. pp. 481–490. IEEE Computer Society, Los Alamitos, CA (1998)
7. Tunstall, B.P.: Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Inst. Technol., Atlanta, GA (1967)
8. Weiner, P.: Linear pattern matching algorithms. In: 14th IEEE Symposium on Switching and Automata Theory. pp. 1–11 (1973)