TCS-TR-A-10-44

TCS Technical Report

An Improvement of STVF Code by Almost Instantaneous Encoding

by

Takashi Uemura Satoshi Yoshida Takuya Kida

Division of Computer Science

Report Series A

May 17, 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

An Improvement of STVF Code by Almost Instantaneous Encoding

Takashi Uemura Satoshi Yoshida Takuya Kida *

May 17, 2010

Abstract

An improved STVF coding algorithm is presented in this paper. STVF code is a novel variable-length-to-fixed-length code (VF code) in which the parse tree is constructed by pruning a suffix tree for an input text. It gives a better compression ratio rather than the previous VF codes, and it is a promising scheme for compressed pattern matching and applications to large-scale text databases. However, its compression ratio is still worse than that of a state-of-the-art compression method. The improvement of the compression ratio of STVF code is required. In our method, all the incomplete inner nodes in addition to leaves in the parse tree are assigned codewords, while just leaves are assigned in the original STVF coding. This code assignment leads an improvement in compression ratio because we can prune the parse tree of infrequent leaves and extend the other frequent edges. In practice experimental results show that the proposed method improves more than 18% in compression ratio for natural language texts in comparison with the original STVF coding.

1 Introduction

STVF code [Kid09, KS09] is a data compression method that is suitable for compressed pattern matching. We present an improvement algorithm of the STVF coding and discuss its performance in this paper.

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. Compression ratio is usually considered as the most important factor in this research area. Many compression methods have been proposed so far (see [Sal06, Say02]). Among them, the Ziv-Lempel family [ZL77, ZL78] 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 electronic mail

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

in recent years has caused the explosion of unformatted text data, data compression becomes more and more important as the foundations of textual databases for such enormous text data.

For large-scale textual databases, however, a state-of-the-art compression method is unsuitable. Although well-known compression tools like gzip and bzip2, which are based on the state-of-the-art algorithms, can compress text data extremely, they encode with variable-length codewords and the encoded texts are highly complicated. Therefore, it is difficult for users to search or modify a part of data without decompressing. An efficient compression method so that we can reuse easily the compressed text data is desired.

On the other hand, from the theoretical viewpoint, an idea of merging search and compression is emerged in the early 90's. The *compressed pattern 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 for this problem takes $O(m + n + R)$ in the worst-case time, where R is the number of the occurrences. However, this combined requirement of searching and compressing is not easy to achieve together, as the only solution before the 90's was to process queries by decompressing the texts and then searching into them.

From the late 90's to the beginning of 2000, several methods that achieve both requirements were appeared [SMT+00, RTT02]. Surprisingly, they could improve the search speed in almost linear to the compression ratio, namely, they can perform pattern matching on compressed texts faster than an ordinary search algorithm on uncompressed texts. This broke out of the paradigm, as we can use data compression to make pattern matching fast [TSM+01], and gives a new criterion of adopting a compression method.

The key point for the achievement is to select a compression method that is suitable for pattern matching. 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 a code whose length is constant times of bytes is better.
- It has a static and/or compact dictionary.

Some variable-length-to-fixed-length codes (*VF codes* for short) have these features. A VF code is a coding scheme that assign fixed length codewords to variable length substrings in the original text. Well-known compression methods like Huffman codes, arithmetic codes, and LZ-families, are based on variable-length codewords, and unsuitable for compressed pattern matching. In practical applications, a compression

method that achieves a good compression ratio with preserving the above features, is strongly desired. However, these features are disadvantages in compressing data. There has long been no VF codes for practical use in fact.

Recently, Maruyama *et al.* [MTST08] presented an excellent compression method named as BPEX for compressed pattern matching, which is a variation of Byte-Pair-Encoding [Gag94] (BPE for short) that is a kind of VF codes. In BPE scheme, the encoding procedure scans the input text many times and repeats conversion of a frequent pair of bytes (characters) into an unused codeword. Although BPEX achieves a good compression ratio comparable to gzip, its compression speed is quite slow.

Klein and Shapira [KS09] and Kida [Kid09] presented independently a VF code based on suffix tree (STVF[1] for short). In STVF code, a frequency-base-pruned suffix tree is used as a parse tree. The compression ratio of STVF code is better than the classical VF codes like Tunstall code [Tun67]. Although the compression speed is faster than that of BPEX, its compression ratio is worse. Therefore, our aim is to improve STVF code in compression ratio without the sacrifices of compression/decompression speed.

A general idea of improvement for VF codes is to assign a codeword to a longer and/or more frequent substring of the input text. In STVF code, some codewords can be assigned to shorter and less frequent substrings, because the parse tree must be constructed so that any internal nodes are complete, namely, for any internal nodes u in the parse tree all the children of u in the suffix tree are contained in the parse tree. If we can assign codewords to the internal nodes, we can prune such useless leaves from the parse tree.

In the proposed algorithm, we adopt an almost-instantaneous encoding strategy, which enables us to assign codewords to the internal nodes in the parse tree. This idea is based on AIVF code proposed by Yamamoto and Yokoo [YY01]. Here we also adopt a method that chooses nodes one by one in descending order of their frequencies from the suffix tree and added them to the parse tree.

The rest of this paper is organized as follows. In Section 2, we make a brief sketch of the original STVF code. In Section 3, we introduce our method. In Section 4, we show the experimental results and discuss about them. Finally, we conclude this paper in Section 5.

2 Preliminaries

2.1 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 $|T| = n$. The string of

[1]Strictly, the methods of [Kid09] and [KS09] are slightly different in detail.

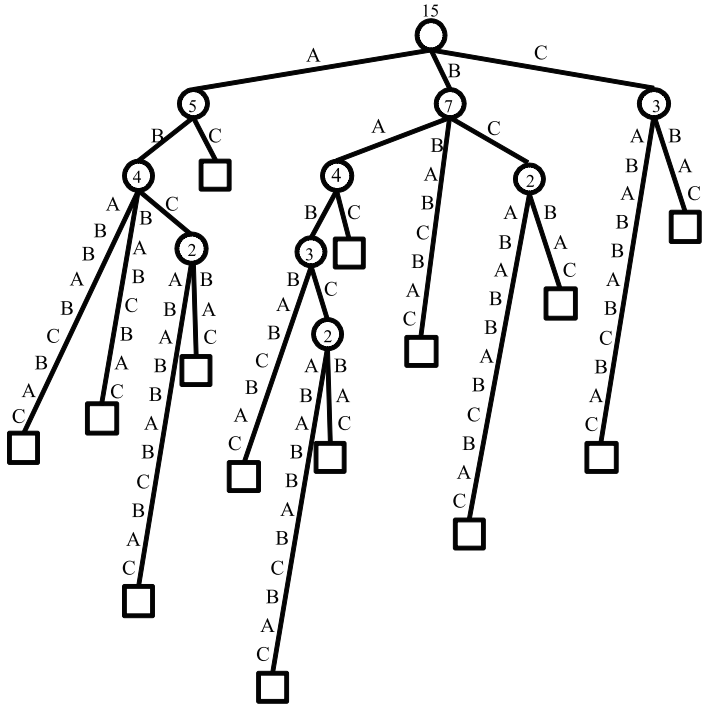

Figure 1: Suffix tree for $T = \mathrm{BABCABABBABCBAC}$.
The squares represent leaves. The circles represent internal nodes and the numbers in the circles are their frequencies.

length zero, denoted by ε, is called the *empty string*. Let T be a string in Σ^*. Then, strings x, y and z are called a *prefix*, *substring*, and *suffix* of T if $T = xyz$, respectively.

2.2 Suffix Tree

For a given text T, the *suffix tree* $ST(T)$ of T is a compacted trie which represents all the suffixes of T, as shown in Fig. 1. 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.
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 suffix of T is represented by a node in $ST(T)$.

For a node v in $ST(T)$ and a symbol $c \in \Sigma$, the function *child* returns the child of v whose label of the ingoing edge starts with c.

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 the frequency is equal to the number of

Algorithm STVF(T, k):
Input: A text T and the length l 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 = {root};
4: while |U| < 2^k do
5:    v =argmax_{v∈U} f(v);
6:    U = U \ {v};
7:    for each child w of v do
8:       U = U ∪ {w};
9:       if v is a leaf of ST(T) then
10:         truncate the label of w to length 1;
11:      add w to 𝒯;
12:   end for
13: end while
14: assign codewords to the elements in U;
15: return 𝒯;
```

Figure 2: Algorithm of constructing a parse tree of STVF code.

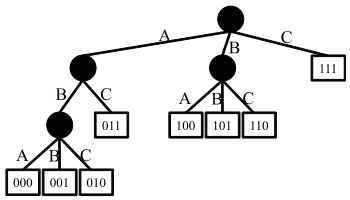

Figure 3: Parse tree of STVF code for T = BABCABABBABCBAC.
The squares represent leaves. The circles represent internal nodes and the numbers in the circles are their frequencies.

leaves in the subtree rooted at v, we can compute all the frequencies of nodes in a post-order traversal. Note that the suffix tree for T can be constructed in linear time to the length of T.

2.3 STVF code

Figure 2 is the parse tree construction algorithm of STVF code, and Fig. 3 shows the parse tree 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 11), the most frequent node v among the leaves in the temporal $\mathcal{T}$ is selected, and then all the children of v in $ST(T)$ are added to $\mathcal{T}$, where all the labels for leaves in $ST(T)$ are truncated to length one. After the above expansion steps, the algorithm assigns codewords to all the leaves in

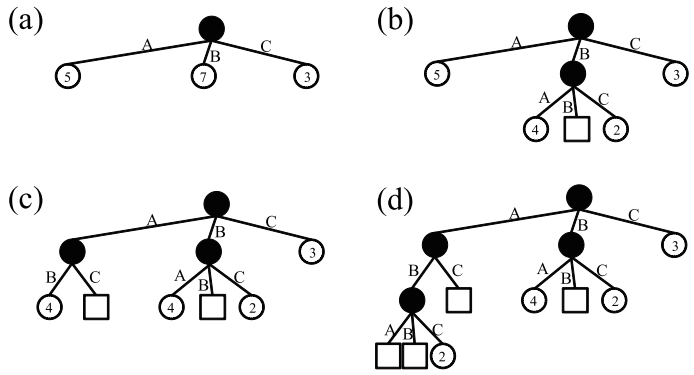

Figure 4: The first four iterations of constructing process of the parse tree
The black circles represent the internal nodes. Only leaves are assigned codewords.

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
```

Figure 5: Encoding algorithm of STVF code.

a left-to-right manner. The first four iterations of the constructing process for the running example is shown in Fig. 4. This construction strategy is similar to that of Tunstall code.

Once the parse tree is constructed, encoding and decoding are simple. The encoding and the decoding algorithms are shown in Fig. 5 and Fig. 6, 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 100110000101010100111.

When we store the compressed data, we have to encode the parse trees together. Here we divide the parse tree into two components: the tree structure and labels on it. The tree structure is encoded by balanced parenthesis [Mun01]. Thus the encoded size for the tree of M nodes is $2M$ bits. For the set of labels, we store by a simple way: enumerate pairs of the label length and the label string and then attach to the compressed text. 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 number of labels.

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

1: **for each** $i \in \{0, \ldots, |C| - 1\}$ **do**
2: $v =$ the node such that $code(v) = C[i]$;
3: **output** $str(v)$;
4: **end for**

Figure 6: Decoding algorithm of STVF code

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** $str(v) \cdot T[i]$ is represented by $\mathcal{T}$ **do**
5: $v =$the node that represents $str(v) \cdot T[i]$;
6: $i = i + 1$;
7: **end while**
8: **output** the codeword assigned to v;
9: **end while**

Figure 7: Modified encoding algorithm.

3 The proposed method

In this section, we introduce our algorithm.

First, we modify the encoding algorithm as in Fig. 7. The algorithm traverses the parse tree while it can move by the character read 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 say it as *the almost-instantaneous encoding*.

Next, we introduce the algorithm for constructing a parse tree as in Fig. 8. The basic idea of the algorithm is to choose a node from the suffix tree, which is the most frequent node that 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 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 algorithm of constructing the parse tree. For a given text T, we first construct the suffix tree $ST(T)$ of T and truncate the labels of the leaves in

Algorithm ImprovedSTVF(T, k):
Input: A text T and the length l 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 | child of the root of ST(T)};
4: for each v ∈ V do
5:    add v to 𝒯;
6:    if v is a leaf then
7:       truncate label(v) to length 1;
8:    U = U ∪ v;
9:    V = (V \ {v}) ∪ {w | w is a child of v};
10: end for
11: while |U| < 2^k do
12:    v =argmax_{v∈V} f(v);
13:    add v to 𝒯;
14:    U = U ∪ v;
15:    V = (V \ {v}) ∪ {w | w is a child of v};
16:    p = v's parent;
17:    if #{v ∈ V | w is a child of p} = 1 and p's parent ≠ root then
18:       w = p's only child in V;
19:       U = (U \ {p}) ∪ {w};
20:       V = (V \ {w}) ∪ {x | x is a child of w};
21:    end if
22: end while
23: assign codewords to the elements in U;
24: return 𝒯;
```

Figure 8: Construction Algorithm for Parse Trees.

$ST(T)$ to length one. Let V be the set of nodes in $ST(T)$ and U also be the set of nodes that will be in the parse tree. Next, to ensure the algorithm can encode the text correctly, we add the root and all the children of the root to U. Then, we repeat the following procedure while $|U| < 2^k$: we first choose the node v whose frequency is maximal in $V \setminus U$ and add it to U. If there is only one child $w \in V$ of the node p, which is also the parent of v, is not in U, we add w to U and remove p from U. An internal node u in the parse tree is *complete* if the parse tree contains all the children of u in the suffix tree. We do not assign a codeword to a complete node because the traversal in the encoding process never fails at any complete nodes. The node p is now complete and thus it is not assigned a codeword. Finally, we assign uniquely 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. We can parse the input text to five substrings by using the parse tree, as BABC/AB/AB/BABC/BAC, and encoded to 101000000101110. Note that

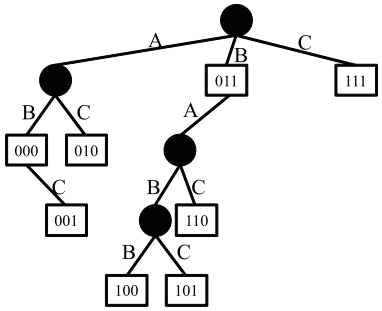

Figure 9: Parse tree of our method for $T = \mathrm{BABCABABBABCBAC}$. The squares represent leaves. The circles represent internal nodes and the numbers in the circles are their frequencies.

the encoded length becomes shorter than that of STVF code in the previous section. Also note that the algorithms of decoding and storing the parse tree are common to STVF expect 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 1 *Let T be a given text and $\mathcal{T}$ be a parse tree constructed by the algorithm in Fig. 8. For any suffix s of T, there exists more than one nodes in $\mathcal{T}$ which represent nonempty prefixes of s and the node which represents the longest prefix of s in $\mathcal{T}$ is assigned a codeword.*

proof. The former is clear because all the children of root are contained in $\mathcal{T}$ and the children will be assigned codewords regardless of whether they are complete or not. We next prove the latter by 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 v. □

4 Experimental Results

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

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

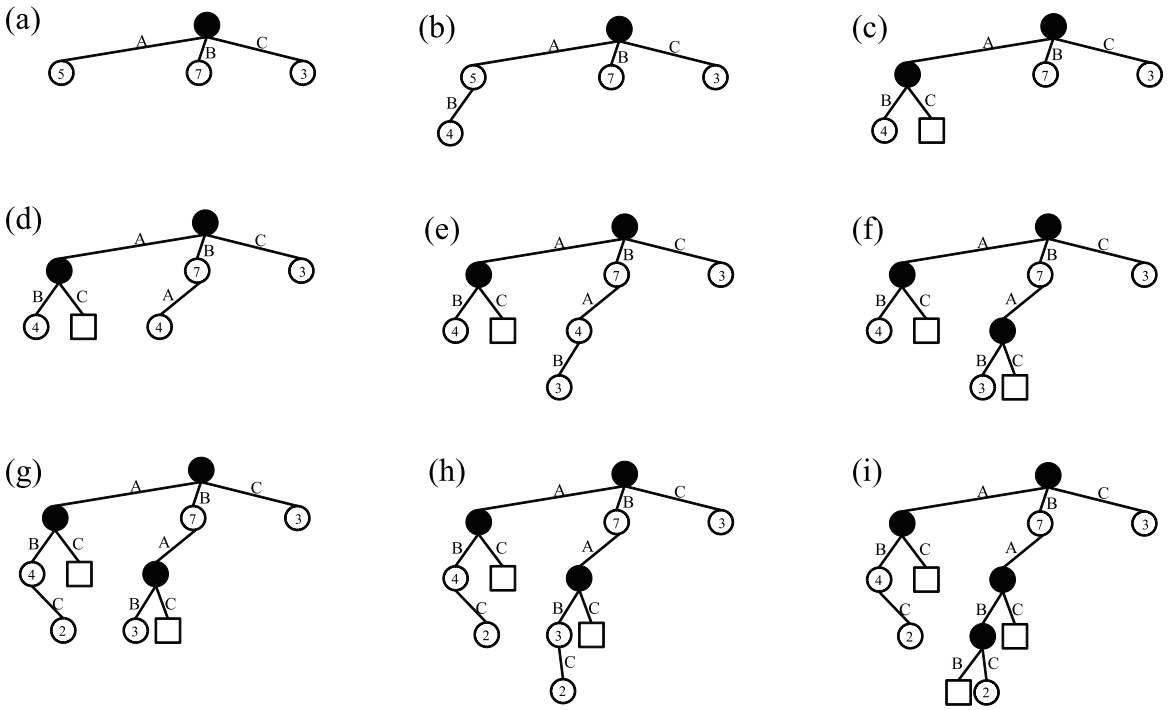

Figure 10: Proposed algorithm for constructing a parse tree
The black circles represent the complete internal nodes, which are not assigned codewords.

Table 1: About text files to be used

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

In order to decide the best length of codewords, we first experimented on compression ratios, compression times, and decompression times against the length l of codewords. In this experiment, we used only bible.txt.

Figure 11 shows the compression ratios. The compression ratio is defined as the following formula: (compressed file size)/(original file size). Proposed algorithm achieved a better performance than the others, especially when the codeword length is short.

Figure 12 compares the compression times. We measured the CPU times by the time command on Linux. As shown in this figure, Tunstall is the fastest.

The results in Fig. 13 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 $l = 16$. Thus, we set the codeword length to 16 in the following experiment.

Next, we have compared five compression algorithms: Tunstall [Tun67], STVF [Kid09], the proposed method (Proposed for short), BPEX [MTST08], and bzip2.

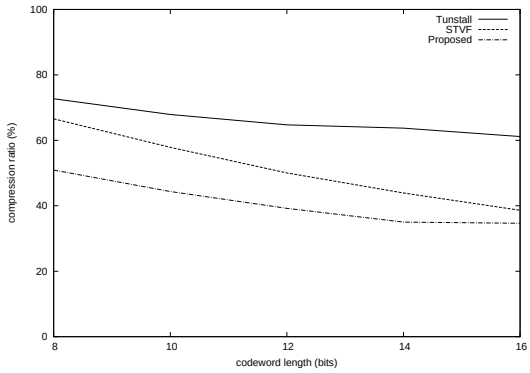

Figure 11: Compression ratio against codeword length.

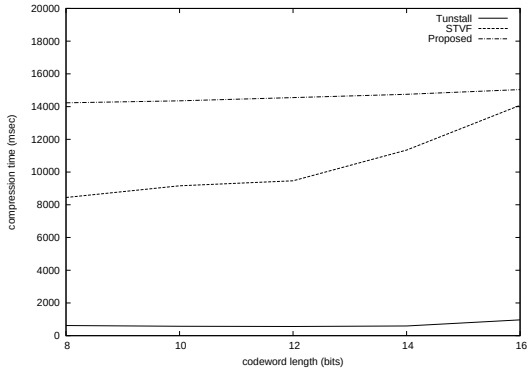

Figure 12: Compression time against codeword length.

bzip2 is a widely used and one of the state-of-the-art compression tools. Thus, we include bzip2 as a reference. We used the three texts in Table 1 as test data.

The compression ratios are shown in Table 2. The proposed method improves the compression ratio approximately by 18% on bible.txt, and by 12% on world192.txt in comparison with the original STVF. Tunstall compresses most effectively on E.coli. BPEX totally achieves a high compression ratio among VF codes.

The compression times are shown in Table 3. STVF and Proposed are two times faster than BPEX. However, compared with Tunstall, STVF and Proposed 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. Proposed and STVF take less time than the others in almost all the cases.

5 Conclusion

We proposed an improvement algorithm of STVF code and carried out several experiments for evaluating its performance. The experimental results showed that it

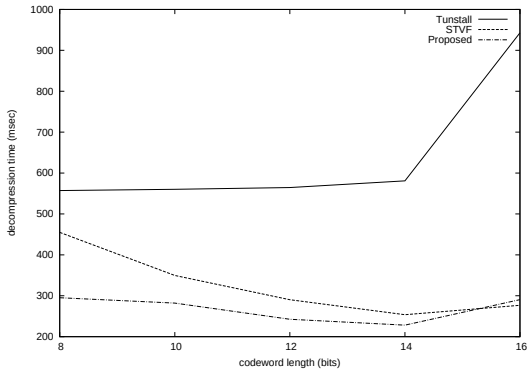

Figure 13: Decompression time against codeword length.

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%
Proposed	34.67%	43.97%	28.89%
BPEX	28.05%	26.58%	28.72%
bzip2	20.89%	19.79%	26.97%

improves in compression ratio for a natural language text at almost no expense of compression speed. They also showed that the compression speed of our method is faster than that of BPEX and the decompression speed is faster than that of bzip2 and comparative to BPEX. However, it revealed that the compression ratio of our method does not reach yet to the level of a state-of-the-art compression and the compression speed is rather slow.

We have to reduce the cost of constructing the parse tree for speeding up compression process. In the STVF coding and the proposed method, we construct the suffix tree for the whole input text at once to construct the parse tree, which takes much time and space. One of the improvement ideas is to prune the suffix tree dynamically while constructing it. We have already implemented a part of this idea and obtained some good results.

To improve the compression ratio, we also have to develop a succinct representation of the parse tree, which contributes to the total compression size. Moreover, we can consider that we adopt the idea of BPEX into STVF codings. These are our future works.

Acknowledgements

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

Table 3: Compression times (msec)

method	bible.txt	world192.txt	E.coli
Tunstall	965	651	7718
STVF	14185	9398	9028
Proposed	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
Proposed	291	223	320
BPEX	335	209	361
bzip2	537	361	794

References

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

[Gag94] Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

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

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

[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)*, volume 5280 of *LNCS*, pages 27–38, Nov. 2008.

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

[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.

[Sal06] David Salomon. *Data Compression: The Complete Reference*. Springer, 4th edition, 2006.

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

[SMT+00] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, 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.

[TSM+01] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Speeding up string pattern matching by text compression: The dawn of a new era. *Transactions of Information Processing Society of Japan*, 42(3):370–384, 2001.

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

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

[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.