TCS-TR-A-10-48

TCS Technical Report

On Performance of Compressed Pattern Matching on VF Codes

by

SATOSHI YOSHIDA AND TAKUYA KIDA

Division of Computer Science

Report Series A

November 19, 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

On Performance of Compressed Pattern Matching on VF Codes

Satoshi Yoshida Takuya Kida[*]

November 19, 2010

Abstract

This paper discusses about pattern matching on Variable-to-Fixed-length codes (VF codes). A VF code is a coding scheme whose codeword lengths are fixed, and thus it is suitable for comprssed pattern matching. However, there are few reports showing its efficiency so far. We present some experimental results about the compression ratios and encoding/decoding speeds besides pattern matching performance on Tunstall codes and STVF codes. We also present how an entropy coding affects to VF codes.

1 Introduction

We have addressed speeding up of pattern matching on texts by designing a compression method that is suitable for compressed pattern matching. 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. Although compressing and searching were hard to go together, from the late 90's to the beginning of 2000, the compression methods which achieve high search performance were appeared [SMT$^+$00, SdMNZBY00, RTT02]. This broke out of the paradigm, as we can use data compression to make pattern matching fast, and gives a new criterion of adopting a compression method [TSM$^+$01]. Since then, several compression methods which are suitable for compressed pattern matching have been proposed [BINP03, BFNE03, MTST08, KBN08, KS09, BFL$^+$10].

We have paid attention to *Variable-to-Fixed-length codes* (VF codes for short) and improved them against such a background [Kid09, YK10]. A VF code is a source coding that assigns fixed length codewords to variable length substrings in an input text; all codeword boundaries are obvious and thus VF codes are suitable for pattern matching. Since existent VF codes have poor compression ratios, they are paid less

[*]Graduate School of Information Science and Technology, Hokkaido University. Kita 14-jo, Nishi 9-chome, Kita-ku, 060-0814 Sapporo, Japan. {syoshid,kida}@ist.hokudai.ac.jp.

attentions in spite of having preferable engineering aspect that all codewords are the same length. If we could tolerate taking time long to compress, VF codes achieve to the level of gzip in compression ratios [UYK^{+}10]. On the other hand, to the author's knowledge so far, there are few reports showing the efficiency of pattern matching on VF codes for real texts in actual.

In this paper, we demonstrate the performance of doing pattern matching on VF codes by experiments. We tested two VF codes: *Tunstall code* [Tun67] and *STVF code* [Kid09]. Tunstall code, which is a classical and typical VF code, is proved to be a optimal VF code for a memory-less information source (see [Sav98]); its average of codeword length par symbol comes asymptotically close to the entropy of the source when the code length goes to infinity. STVF code is a VF code proposed by Kida [Kid09], which utilizes a frequency-based-pruned suffix tree as a dictionary[1]. Suffix tree is a well-known index structure that stores all substrings in the target text compactly. STVF code has better compression ratios than Tunstall code and Huffman code.

Our experimental results showed that doing pattern matching on Tunstall codes directly for natural language texts is $1.3 \sim 2.4$ times faster than the decompressing and then searching method on gzipped texts by Unix command *zgrep*. For DNA data, it is over $6 \sim 15$ times faster than zgrep. On the other hand, doing pattern matching on STVF codes directly for natural language texts is $1.3 \sim 2.0$ times slower than zgrep.

For improving compression ratio, to combine VF codes and an entropy encoding like Huffman code is a natural idea. We have also investigated on changes in performance of VF codes with entropy encodings; we experimented combinations of VF codes and entropy codings and measured their compression ratios, compressing/decompressing times, and pattern matching speeds. The experiment results showed that entropy codings depress the pattern matching speeds by the decoding time but improve the compression ratios for natural language texts by over $20\% \sim 40\%$ for Tunstall codes, and over $24\% \sim 30\%$ for STVF codes.

2 Preliminaries

2.1 VF 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.

[1]Klein and Shapira [KS09] previously proposed a VF code, called DynC, based on the same idea, but the encoding algorithm is slightly different from [Kid09].

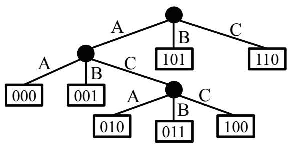

Figure 1: An example of a parse tree.
The squares represent leaves, where codewords are assigned. The black circles represent internal nodes.

Consider that we encode an input text $T \in \Sigma^*$ by a VF code of length ℓ-bits codewords. Assume that a parse tree $\mathcal{T}$ that has t leaves is given, and each leaf in $\mathcal{T}$ is numbered as a ℓ-bits integer, where $t \leq 2^\ell$. 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.

2.1.1 Tunstall code

Tunstall code [Tun67] is an optimal VF code (see also [Sav98]) 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^ℓ

(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 s be the number of internal nodes in $\mathcal{T}^*$. Since the number of leaves in $\mathcal{T}^*$ equals to $s(k-1)+1$, which is less than or equal to 2^ℓ. Hence, $s = \lfloor (2^\ell - 1)/(k-1) \rfloor$.

2.1.2 STVF code

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

Now consider to code T with codeword length ℓ. The pruning algorithm is as follows: (i) Let a tree that consists of the root of $ST_P(T\$)$ and its direct children, be the first parse tree candidate $\mathcal{T}'_1$; (ii) Select a node v which has the highest frequency among all leaves in $\mathcal{T}'_i$; Here let L_i be the number of leaves in $\mathcal{T}'_i$ and C_v be the number of direct children of v; (iii) Add all children in $ST_P(T\$)$ of v to $\mathcal{T}'_i$ as leaves if $L_i + C_v - 1 \leq 2^\ell$, and let it be $\mathcal{T}'_{i+1}$ for the new candidate, otherwise stop the pruning; If child u of v is a leaf in $ST_P(T\$)$, cut the label on the edge from v to u so that the label length becomes 1, leaving the first character on the edge; (iv) Repeat Step (ii) and (iii). The encoding/decoding procedures are as the same as those of Tunstall code.

2.2 Entropy encoding

Shannon's source coding theorem says essentially that a message of n symbols can be compressed to nH bits on average (but not further) for a text of its entropy H, and also says that almost optimal codings exist, which are called *entropy encodings*.

Huffman coding and arithmetic coding [RL79] are the most common entropy encodings. Arithmetic coding is a coding that codes the cumulative frequency of the entire message into bits, while Huffman coding encodes each character into a bit sequence according to its frequency. Therefore, arithmetic coding usually approaches to the entropy rate much closer than Huffman coding does. There are, however, two problems on the original arithmetic coding. One is that we have to calculate real number operations in arbitary accuracy, and the other is that such real number operations take much time for compressing and decompressing. To avoid these problems, Range coder[Mar79], a kind of arithmetic coding, simulates the real number operations with

[2]Strictly, the methods of [Kid09] and [KS09] are slightly different in detail. However, we call them the same name here since the key idea is the same.

those of fixed-lenth integers by some way. Range coder is adopted as a high speed entropy encoder by many compression methods.

It is common idea to do entropy coding after translating the source input into some forms. It would be also effective in compressing to combine an entropy coding with VF codes. As stated above, the output of VF code is a sequence of ℓ-bits integers, where each codeword is corresponding to a variable-length substring, called block. Therefore, to encode it by an entropy coding means to do block encoding, where each block is parsed by the VF coding. It means only that the number of different blocks is up to 2^ℓ. Although the number of useless codewords increases when ℓ becomes larger since the occurrence frequency for each codeword in the medium sequence decreases, entropy codings can capture the bias of the occurrence. Note that we need to store the dictionary for entropy coding in addition to the parse tree for VF codes. For example, the cost of storing the dictionary for Huffman coding increases according to 2^ℓ.

2.3 Pattern matching on VF codes

Consider that a text $T = T[1:u]$ is coded by a VF code into a compressed text Z. Note that Z is a sequence of blocks $Z = b_1, b_2, \ldots, b_n$, where each block $b_i (1 \leq i \leq n)$ is represented by a ℓ-bits integer and corresponding to a factor of T. We denote by $w(b_i)$ the string represented by b_i. We define the compressed pattern matching problem on VF codes as follows:

Definition 1 *Given: a pattern $P = P[1:m]$ and a VF coded text $Z = b_1, b_2, \ldots, b_n$. Find: all locations at which P occurs in the original text $T[1:u] = w(b_1)w(b_2)\cdots w(b_n)$ without decompressing it.*

Although we omit the detail discussion, this problem can be solved in $O(n+R)$ time after an $O(|D|+m^2)$ time preprocessing using $O(|D|+m^2)$ space, where m, n, R, and $|D|$ are the pattern length, compressed text length, the number of pattern occurrences, and the size of the parse tree, respectively. The proof is done by Kida *et al.*[KST$^+$99]; they proposed a general framework for compressed pattern matching, named *collage system*. Collage system is a formal system that captures compressed texts encoded by dictionary-based compression methods. They also present a general pattern matching algorithm on collage systems. VF codes can be framed into (regular) collage systems; namely, a pattern matching algorithm on VF codes is systematically delivered from the general algorithm.

3 Experimental Results

In our experiments, we tested Tunstall coding and STVF coding for VF codings, and tested Huffman coding and Range coding for entropy codings. The codewords of

Table 1: Text files for experiments

Texts	size(byte)	$\|\Sigma\|$	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)

VF codes are integers of the range of 0 to $2^\ell - 1$. Although both Huffman coding and Range coding are theoretically independent from the source alphabet size, most the programs that are available on the Internet assume that an input is coded with ASCII or bytewise codes. We made maximum efforts, but we could not find proper programs. Therefore, in this time, we newly implemented Huffman coding and Range coding which can deal with a large alphabet over 256.

We have implemented Tunstall coding, STVF coding, Huffman coding, and Range coding. We abbreviate them as Tunstall, STVF, Huf and RC respectively. We denote the combination of two compression method as concatenation of the abbreviations with "+" between them, such as STVF+Huf, Tunstall+RC, and so on. We also indicate the codeword length of Tunstall and STVF after them in parentheses, such as STVF+Huf(8), Tunstall+RC(16), and so on.

All programs we have implemented are written in C++ and compiled by g++ of GNU, version 3.4. We ran our experiments on an Intel Xeon processor of 3.00GHz dual core hyper threading with 12GB of RAM running on Red Hat Enterprise Linux ES release 4. We used E.coli, bible.txt, and world192.txt as test corpora, which are selected from "the Canterbury corpus[3]." We also used dazai.txt, the collection of japanese novel texts written by Osamu Dazai, from Japanese corpus "J-TEXTS[4]." The file dazai.txt is encoded with UTF-8. For the details, please refer Table 1.

At first, we show the results of compression performance. The results of compression ratios are shown in Fig. 2. We have compared the compression ratios, compression/decompression times of STVF, STVF+RC, STVF+Huf, Tunstall, Tunstall+RC, and Tunstall+Huf, setting the codeword length 8 and 16. We have added the results of gzip for reference. We measured (compressed file size)/(original file size) as the compression ratios. As seen from Fig. 2, the compression ratios of STVF are usually better than those of Tunstall. However, after applying Huf or RC, compression ratios of them become into almost the same level. In STVF, there are a few or no improvement by Huf and RC when the codeword length is 16.

The results of compressing times are shown in Fig. 3. We measured CPU times by time command on Linux. As seen from Fig. 3, the compression does not become slower so much when the codeword length is 8, nevertheless we applied Huf/RC after Tunstall/STVF. STVF+Huf and Tunstall+Huf become slower when the codeword

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

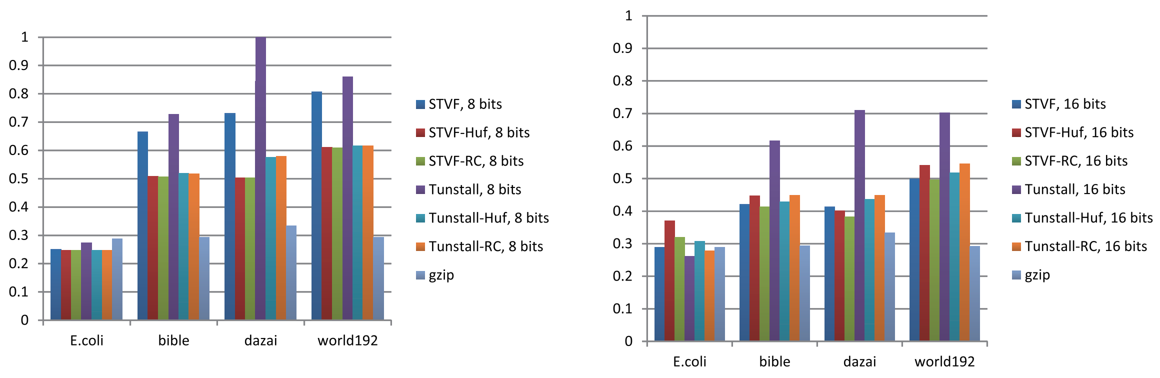

Figure 2: Compression ratios.

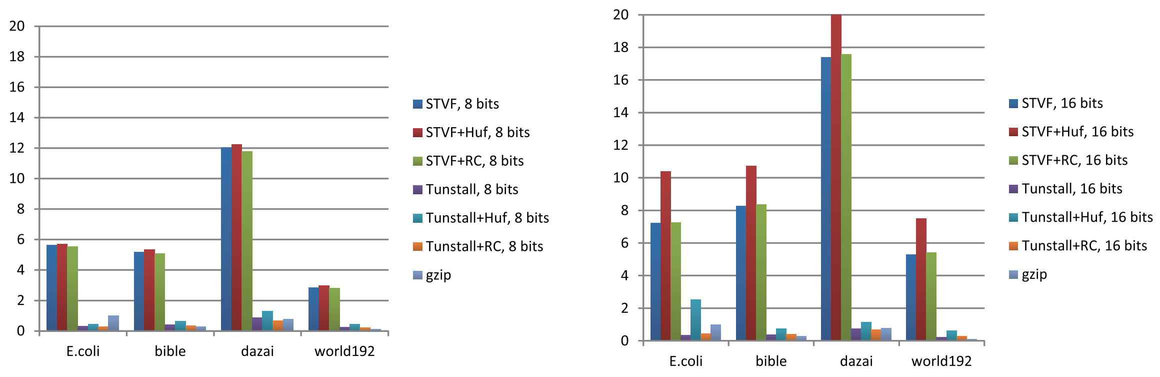

Figure 3: Compression times.

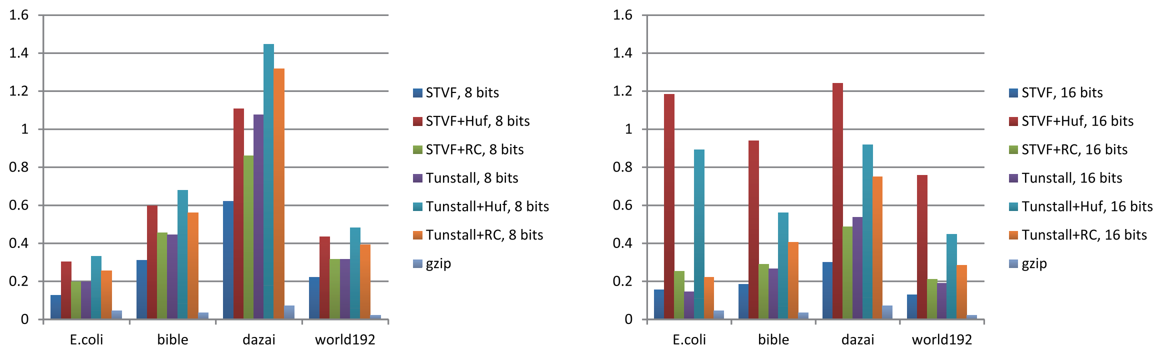

Figure 4: Decompression times.

length is 16, whereas STVF+RC and Tunstall+RC do not become slower than STVF and Tunstall, respectively.

The results of decompressing times are shown in Fig. 4. As seen from Fig. 4, the decompression of STVF is faster than that of Tunstall. Applying RC/Huf after STVF/Tunstall makes their decompressing time slower by the decoding time for RC/Huf. The decompression of Tunstall+Huf and STVF+Huf are faster than those of Tunstall+RC and STVF+RC, respectively. The difference between them is tend to be larger when the codeword length is longer.

Next, we show the resutls of pattern matching performance. We have compared

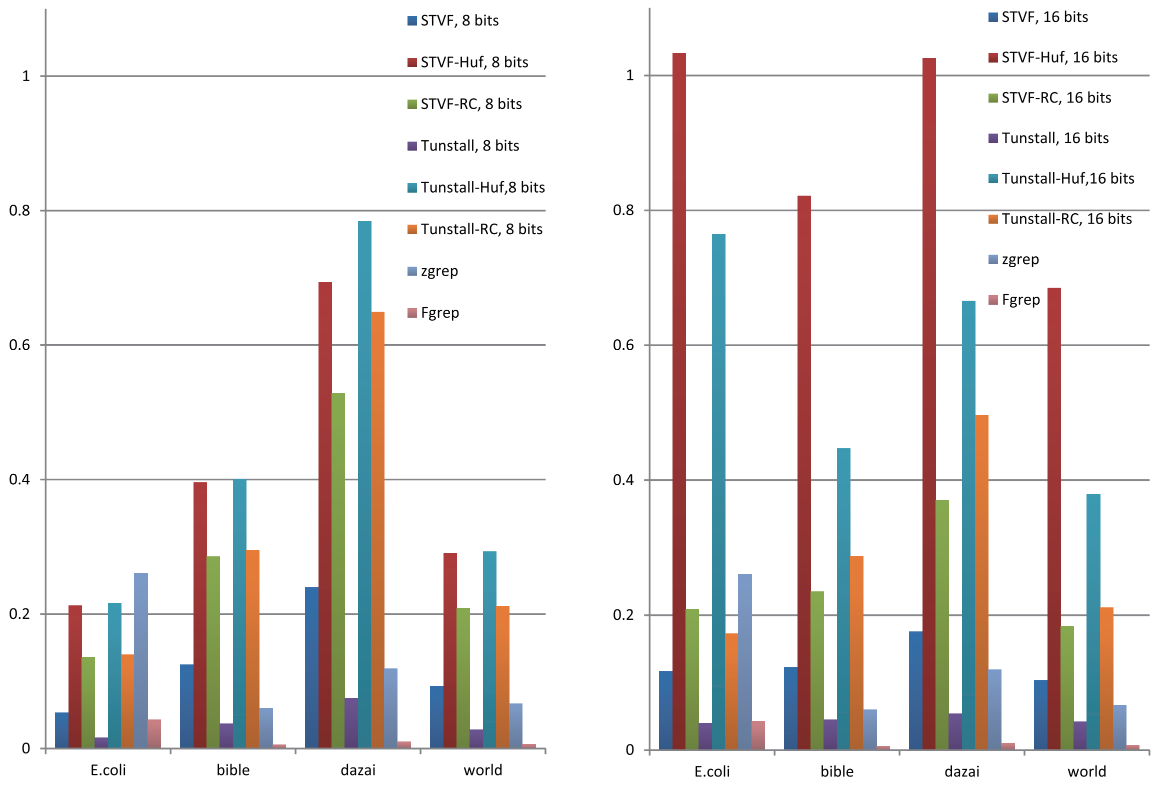

Figure 5: Pattern matching times.

pattern matching performance of Tunstall, STVF, Tunstall+RC, STVF+RC, Tunstall+Huf and STVF+Huf. We have added the results of zgrep on gzipped texts and fgrep on uncompressed texts for reference. In this time, we have implemented the pattern matching algorithms on VF codes whose preprocessing time is $O(m^3)$, not $O(m^2)$, for the convenience of implementation. We collected the patterns from the text at random position. The lengths of patterns are 5 to 50; we collected 50 patterns for each length.

First, we compare the pattern matching times on patterns of length 15. The results are shown in Fig. 5. The pattern matching performance becomes worse if we apply RC/Huf to Tunstall/STVF. As seen from Fig. 5, the pattern matching of Tunstall+Huf and STVF+Huf are slower than that of Tunstall+RC and STVF+RC, respectively. STVF, STVF+RC, Tunstall, and Tunstall+RC are faster than zgrep for E.coli. Only Tunstall(8) is faster than zgrep in any cases.

Finally, we compare the preprocessing times and text scanning times for pattern matching. We measured them by embeding instructions in the source codes to obtain the CPU time. The results of the preprocessing times and text scanning times are shown in Fig. 6 and Fig. 7, respectively. Although the preprocessing times for Tunstall and STVF are almost the same, Tunstall/STVF(16) are slower than Tunstall/STVF(8). As seen from Fig. 6, they increase quickly when pattern length is longer, since we employed an $O(m^3)$ implementation. On the other hand, the scanning times are almost constant. Namely, the pattern length does not affect on the scanning time. Tunstall is usually faster than STVF. Moreover, scanning of Tunstall/STVF(16) are faster than those of Tunstall/STVF(8), respectively.

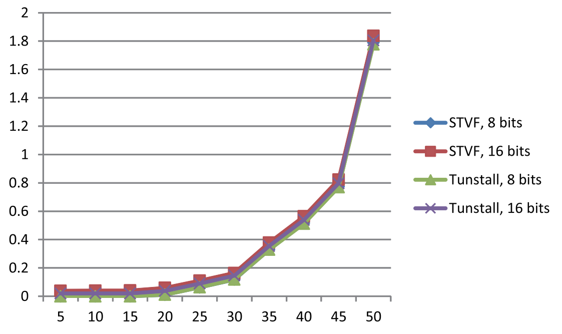

Figure 6: Preprocessing times for pattern matching.

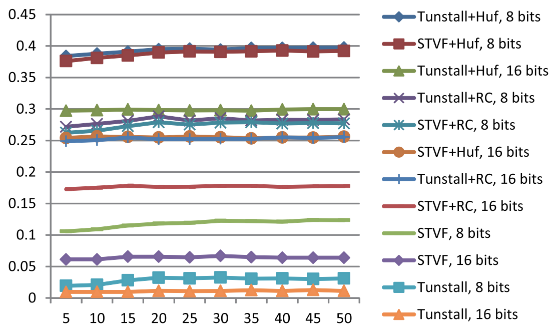

Figure 7: Text scanning times.

4 Conclusions

In this paper, we demonstrated the performance of pattern matching on VF codes by experiments. When the codeword length is longer, scanning encoded texts becomes faster, while preprocessing takes much more time since the parse tree becomes larger. When the pattern length is shorter enough, about shorter than 20, it also revealed that compressed pattern matching on VF codes was faster than zgrep.

Doing entropy coding to VF codes not only improves compression ratios, but makes almost no loss in compression/decompression time; unexpectedly, there are situations that entropy coding can reduce them. The reason is considered to be that the total I/O time decrease by the reduction in the amount of data stored into a hard disk. On the other hand, the pattern matching speeds depress by the time needed for doing entropy decoding.

Since we employed an implementation whose preprocessing for pattern matching takes $O(m^3)$ time, the pattern matching speed becomes worse than that on the uncompressed texts when the pattern is longer than about 20. In any cases, Unix command fgrep was the fastest. To do pattern matching faster, it is essential to improve the pattern matching algorithm itself besides its efficient implementation. The algorithm we realized in this time was a prefix-type algorithm, but a suffix-type algorithm on collage systems have also been proposed [SMT$^+$00]. To implement it for VF codes is one of our future works. From the lack of time, we could not compare our methods to the modern compression methods, such as Dense codings [BINP03, BFNE03], BPEX [MTST08], and so on. It is also our future work.

References

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

[BFL+10] Nieves R. Brisaboa, Antonio Fariña, Juan-Ramón López, Gonzalo Navarro, and Eduardo R. Lopez. A new searchable variable-to-variable compressor. In *DCC*, pages 199–208, 2010.

[BFNE03] Nieves R. Brisaboa, Antonio Fariña, Gonzalo Navarro, and María F. Esteller. (s, c)-dense coding: An optimized compression code for natural language text databases. In *SPIRE*, pages 122–136, 2003.

[BINP03] Nieves R. Brisaboa, Eva Lorenzo Iglesias, Gonzalo Navarro, and José R. Paramá. An efficient compression code for text databases. In *ECIR*, pages 468–481, 2003.

[KBN08] Shmuel Tomi Klein and Miri Kopel Ben-Nissan. Using fibonacci compression codes as alternatives to dense codes. In *DCC*, pages 472–481, 2008.

[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.

[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.

[Mar79] G. N. N. Martin. Range encoding: An algorithm for removing redundancy from a digitised message. In *Proc. Video and Data Recording Conference*, pages 24–27, 1979.

[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.

[RL79] J. Rissanen and G.G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[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.

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

[SdMNZBY00] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18:113–139, April 2000.

[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.

[UYK$^+$10] Takashi Uemura, Satoshi Yoshida, Takuya Kida, Tatsuya Asai, and Seishi Okamoto. Training parse trees for efficient VF coding. In *Proc. 17th International Symp. on String Processing and Information Retrieval (SPIRE2010)*, pages 179–184, 2010.

[YK10] Satoshi Yoshida and Takuya Kida. An efficient algorithm for almost instantaneous VF code using multiplexed parse tree. In *Proceedings of the 2010 Data Compression Conference (DCC2010)*, pages 219–228. IEEE Computer Society, 2010.