IPSJ Transactions on Databases Vol.6 No.4 17–23 (Sep. 2013)

Regular Paper

A Variable-length-to-fixed-length Coding Method Using a Re-Pair Algorithm

Satoshi Yoshida[1,a)] Takuya Kida[1,b)]

Received: March 20, 2013, Accepted: May 1, 2013

Abstract: In this study, we address the problem of improving variable-length-to-fixed-length codes (VF codes). A VF code is an encoding scheme that uses a fixed-length code, which provides easy access to compressed data. However, conventional VF codes generally have an inferior compression ratio compared with variable-length codes. A method proposed by Uemura et al. in 2010 delivered a good compression ratio that was comparable with that of gzip, but it was very time consuming. In this study, we propose a new VF coding method that applies a fixed-length code to a set of rules extracted using the Re-Pair algorithm, which was proposed by Larsson and Moffat in 1999. The Re-Pair algorithm is a simple offline grammar-based compression method, which has good compression-ratio performance with moderate compression speed. We also present experimental results, which demonstrates that our proposed coding method is superior to the existing VF coding method.

Keywords: compressed pattern matching, data compression, grammar-compression, VF code

1. Introduction

In this study, our objective is to develop an effective *variable-length-to-fixed-length code* (VF code). A VF code is a coding scheme that parses an input text into a consecutive sequence of substrings, before it assigns a fixed length codeword to each parsed substring. This code allows us to access any block randomly because the codeword boundaries are clear, which is a valuable feature from an engineering viewpoint. For example, VF codes have been evaluated for speeding up the search of compressed texts [8], [14].

Early VF codes [22], [24], [28], [30], which are exemplified by *Tunstall code* [25], have inferior compression ratios in practice compared with other well-known compression tools such as gzip and bzip, which employ variable-length codewords, although the compression ratios of VF codes are theoretically better than those of fixed-length-to-variable-length codes (FV codes) [30]. The effective elimination of data redundancy is difficult for VF code because all of the codewords have equal length. For example, the compression ratio of the Tunstall code is usually 60% or more. Thus, less attention was paid to the practical applications of early VF codes.

The compression ratio of a conventional VF code depends on a parse tree, which is a dictionary tree used for parsing input text. To improve the compression ratios of VF codes, Uemura et al. [27] proposed a method for training the parse tree by scanning the input text repeatedly. This method achieves a good compression ratio compared with gzip but it required an excessive computational time. Indeed, this method is approximately 100 times slower than Tunstall coding. Therefore, an alternative approach is required to achieve both rapid compression and a good compression ratio.

Improvements depend on how rapidly we can construct a better dictionary, i.e., the method of parsing text is an important issue. This issue is common in *grammar-based compression* [11], which is a compression scheme used to translate an input text into a set of grammar rules, before encoding the rules. Each rule corresponds to a substring in the text, i.e., grammar-based compression uses a rule set as a dictionary for coding. With this type of compression scheme, a smaller set of rules gives a better compression ratio. The extraction of the rule set from the text is related to the method used to parse the text. Finding the optimal grammar is an NP-hard problem [6] but several excellent heuristic algorithms have been proposed for its solution [15], [16], [20]. One promising solution is combining these algorithms with VF coding.

In this study, we propose a method for applying fixed-length coding to the rules extracted using the *Re-Pair algorithm*, which was proposed by Larsson and Moffat [15]. The Re-Pair algorithm is a simple offline grammar-based compression algorithm that replaces the most frequent bigrams in an input text iteratively using nonterminal symbols until all of the bigrams are unique. Our method encodes the rules extracted by the Re-Pair algorithm using fixed-length codewords, whereas the original algorithm utilized variable-length codewords to achieve a very good compression ratio. We exploit a simple characteristic of the algorithm to minimize the reduction in the compression ratio compared with the original algorithm, i.e., the minimum output size occurs frequently during the process of repeated bigram replacement. All of the codewords are equal in length in our method so we can

[1] Graduate school of Information Science and Technology, Hokkaido University, Sapporo, Hokkaido 060–0814, Japan
[a)] syoshid@ist.hokudai.ac.jp
[b)] kida@ist.hokudai.ac.jp

easily estimate the final output size for each intermediate rule set produced by the Re-Pair algorithm. Thus, we can obtain the minimum output at a reasonable cost by preserving the best point and rewinding the rule set back to this point.

The performance of our proposed method was demonstrated by evaluation experiments using specific corpora. The experimental results showed that the compression ratio of the proposed method was approximately equal to that of bzip, although it uses fixed-length codewords. The pattern matching performance was also tested using compressed texts and this confirmed that the compressed pattern matching method was faster with our VF code than UNIX zgrep, which is a typical decompress-then-search method, i.e., gunzip-then-grep.

Our contributions can be summarized as follows.

- We developed a new VF coding method with superior compression ratio and compression time compared with existing VF coding methods. The proposed coding method surpassed gzip and approached to bzip2 in terms of the compression ratio. The compression speed of our method was twice as fast as the original Re-Pair algorithm while the decompression speed was comparable to that of gzip.
- We demonstrated experimentally that pattern matching could be performed faster with a text compressed using our method compared with a text compressed using the decompress-then-search method. This is an advantage of VF code compared with other high-compression methods that use variable-length codewords.

Our proposed coding method is based on a general concept. However, it was not obvious whether the method would actually be effective.

Part of this work was presented at *The Data Compression Conference 2013* [29]. This paper is a full version of Ref. [29], and we carried out additional experiments. We have also added an appendix that discusses compressed pattern matching on VF codes.

2. Related Studies

We aimed to develop a data compression scheme, which would allow us to process compressed data with ease. This issue arose from studies of the *compressed pattern matching problem*.

The compressed pattern matching problem was first defined in a study by Amir and Benson [1] as the task of performing string matching in a compressed text without its decompression. Many pattern matching algorithms have been proposed for each specific compression method [10], [18], [19]. However, most of them are no faster than *the decompress-then-search method*.

Practical and effective methods were proposed from late 1990s until the beginning of 2000 [21], [23]. These methods increased the search speed and they had an approximately linear relationship to the compression ratio, i.e., they could perform pattern matching in compressed texts faster than ordinary search algorithms using uncompressed texts.

After 2000, researchers began to develop a new compression method that was suitable for searching. Thus, Brisaboa et al. proposed a series of *Dense Codes* [2], [3], [4], [5]. Dense codes parse an input text using a morphological analysis tool before encoding it with byte-oriented codewords. Klein and Ben-Nissan [13] devised a variation of the Dense Code by using Fibonacci codes for text compression.

For VF codes, Klein and Shapira [14] and Kida [8] independently presented VF codes based on a suffix tree (*STVF code* [*1]). A frequency-base-pruned suffix tree is used as a parse tree in the STVF code. STVF coding is also suitable for searching because it uses a static dictionary and the codeword boundaries are obvious (see the Appendix for compressed pattern matching on VF codes). The compression ratio of the STVF code is superior to that of classical VF codes such as *Tunstall code* [25], but it is still inferior to state-of-the-art compression methods. Some experimental comparisons of Dense Codes, VF codes, and gzip were presented in Ref. [27].

Various practical algorithms have also been developed for grammar-based compression. Bisection [12] is a grammar-based compression algorithm where the grammar belongs to the class of a straight-line program. Algorithms have also been presented for restricted context-free grammars [11], [15], [20]. For example, Re-Pair [15] and Sequitur [20] are particularly useful because of their good compression ratios.

Maruyama et al. [16] presented an excellent compression method based on context-sensitive grammar, known as BPEX [*2]. This method can be viewed as an extension of Byte Pair Encoding (BPE) [7], which is a restricted version of the Re-Pair algorithm. BPEX improves the compression ratio compared with BPE and its pattern matching performance is extremely good. However, the compression speed of BPEX is slow and it is difficult to decode or perform pattern matching directly after the middle of the compressed data because any codeword in BPEX-compressed data depends on the preceding codeword.

3. Re-Pair Algorithm

The Re-Pair algorithm [15] is a simple offline grammar-based compression method, based on context-free grammars (CFGs). Formally, a CFG is represented by a quadruple (Σ, V, σ, R), where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$ is the terminal alphabet, $V = \{a_{|\Sigma|+1}, a_{|\Sigma|+2}, \ldots, a_{|\Sigma|+|V|}\}$ is the non-terminal alphabet, $\sigma \in V$ is the start symbol, and R is a finite relation from V to $(\Sigma \cup V)^*$. Note that Σ and V are disjoint sets.

The CFG constructed by the Re-Pair algorithm consists of rules in which

$$\sigma \Rightarrow \sigma_1\sigma_2\cdots\sigma_m \quad (\sigma = a_{|\Sigma|+|V|}, \forall\sigma_i \in \Sigma \cup V \setminus \{a_{|\Sigma|+|V|}\}),$$
$$a_i \Rightarrow a_j a_k \quad (|\Sigma| + 1 \le i < |\Sigma| + |V|, 1 \le j, k < i),$$

and all the right-hand sides of the rules are unique.

Algorithm 1 shows the Re-Pair algorithm. The algorithm replaces the most frequent bigrams in the sequence with a new non-terminal symbol and adds the replacement into R as a rule. The algorithm repeats this procedure until there are no repeated bigrams, i.e., the frequencies of all bigrams are equal to one. After that, the algorithm adds the start symbol σ, which generates the obtained sequence, into R. Finally, the algorithm encodes all the

[*1] The method of Ref. [14] is referred to as DynC in their paper, where the encoding algorithm is slightly different from that used by Ref. [8].

[*2] "BPEX" is simply the name of the program written by Maruyama but we refer to it as the name of their method.

Algorithm 1 The Re-Pair algorithm.

Input: A text $T = T[1..n]$ and an alphabet $\Sigma = \{a_1, a_2, \cdots, a_{|\Sigma|}\}$.
Output: The binary coded sequence of the rule set R for T.

1: $s \leftarrow |\Sigma| + 1$, $R \leftarrow \emptyset$.
2: Add $(\sigma \Rightarrow T)$ to R. // We identify T with the right-side of σ below.
3: **for all** $i \in \{1, \cdots, |\Sigma|\}$ Add $(\alpha_i \Rightarrow a_i)$ to R.
4: Replace every a_i in T with α_i.
5: **while** the frequency of the most frequent bigram in T is not equal to 1 **do**
6: $(\beta, \gamma) \leftarrow$ the most frequent bigram.
7: Add $(\alpha_s \Rightarrow \beta\gamma)$ to R.
8: Replace all the bigrams $\beta\gamma$ in T with α_s by the left-to-right manner.
9: $s \leftarrow s + 1$.
10: **end while**
11: Encode R with an entropy encoding.

rules except for σ by *chiastic slide method*, and encodes σ with *minimum-redundancy codes* [17], [26].

4. Proposed Method

We can easily encode the rule set generated by the Re-Pair algorithm using a fixed-length code so α_i is coded by a $\lceil \lg s \rceil$-bits integer, where s denotes the number of non-terminal symbols except for the start symbol σ. However, the compression ratio would be worse than the original Re-Pair algorithm.

The concept we apply to improve the compression ratio is based on the observation that adding a new rule does not always improve the ratio. The sequence always becomes shorter by replacing bigrams with a new rule but the rule set becomes larger. Thus, the codeword becomes longer so the final output eventually becomes larger. If we find the best value of s, we can obtain the minimum output in this framework. Note that s increases monotonically by one after each repetition.

The final output is obtained as a sum of encoded rules. For the original Re-Pair algorithm, it is difficult to predetermine whether the output will become shorter prior to replacing bigrams because the algorithm employs a variable-length code.

We can easily estimate the output size using a fixed-length code. Each non-terminal symbol α_i is encoded into a $\lceil \lg s \rceil$-bits integer. We output $s - |\Sigma|$ bigrams in addition to the information of Σ as the dictionary, where the dictionary size is $2(s - |\Sigma|)\lceil \lg s \rceil$ bits, as well as some auxiliary bits for storing Σ and s. The lengths of the auxiliary bits are fixed for the same input text, so we do not need to consider them. The right-hand side of the start symbol σ is encoded using $|\sigma|\lceil \lg s \rceil$ bits as the encoded sequence, where $|\sigma|$ is the length of the right-hand side of σ. Therefore, the estimated output size $f(s)$ is given as follows:

$$f(s) = [2(s - |\Sigma|) + |\sigma_s|] \cdot \lceil \lg s \rceil,$$

where σ_s is the sequence corresponding to the initial symbol with a dictionary size of s. The term $|\Sigma|$ is an invariant factor and $|\sigma_s|$ depends on the number of repetitions, which correspond to the size of the rule set R. This means that f depends only on s. In other words, the value of s controls the final output size.

After computing $f(s)$ for each intermediate rule set, we can find the best value of s for f after all the repetitions are complete. We denote this value s as $\hat{s}$. However, it is not sufficient to compare only the current value of f with the next value after replacement, because the value may fall into a local minimum. Therefore, we have to complete all of the iterations.

Algorithm 2 Proposed method.

Input: A text $T = T[1..n]$ and an alphabet $\Sigma = \{a_1, a_2, \cdots, a_{|\Sigma|}\}$.
Output: The binary coded sequence of the rule set R for T.

1: $s \leftarrow |\Sigma| + 1$, $R \leftarrow \emptyset$.
2: $b \leftarrow \infty$, $\hat{s} \leftarrow s$.
3: Add $(\sigma \Rightarrow T)$ to R. // We identify T with the right-hand side of σ below.
4: **for all** $i \in \{1, \cdots, |\Sigma|\}$ Add $(\alpha_i \Rightarrow a_i)$ to R.
5: Replace every a_i in T with α_i.
6: **while** the frequency of the most frequent bigram in T is not equal to 1 **do**
7: $(\beta, \gamma) \leftarrow$ the most frequent bigram.
8: Add $(\alpha_s \Rightarrow \beta\gamma)$ to R.
9: Replace all of the bigrams $\beta\gamma$ in T with α_s from left to right.
10: **if** $f(s) < b$ **then**
11: $b \leftarrow f(s)$.
12: $\hat{s} \leftarrow s$.
13: **end if**
14: $s \leftarrow s + 1$.
15: **end while**
16: **Output** $\hat{s}$ and the information related to Σ.
17: **for** $i \leftarrow |\Sigma| + 1$ **to** $\hat{s}$ **do**
18: **Output** $R(i)$ with $\lceil \lg \hat{s} \rceil$ bits for each symbol.
19: **end for**
20: **for** $i \leftarrow 1$ **to** the size of the right-hand side of σ **do**
21: **Call** procedure REWIND-OUTPUT($\sigma[i]$, $\hat{s}$, R).
22: **end for**

23: **procedure** REWIND-OUTPUT(s, $\hat{s}$, R)
24: **if** $s \leq \hat{s}$ **then**
25: **Output** s with $\lceil \lg \hat{s} \rceil$ bits.
26: **else**
27: $(\beta, \gamma) \leftarrow R(s)$.
28: **Call** procedure REWIND-OUTPUT(β, $\hat{s}$, R).
29: **Call** procedure REWIND-OUTPUT(γ, $\hat{s}$, R).
30: **end if**
31: **end procedure**

There are two approaches for outputting σ with $\alpha_1, \cdots, \alpha_{\hat{s}}$ after obtaining $\hat{s}$. The first approach is to rewind the rule set constructed using the Re-Pair algorithm to produce the intermediate set for $\hat{s}$ and replace T. The second approach is to preserve s and T while the current minimum value of f is updated during repetitions. The first approach can reduce the memory consumption required for encoding but we need to expand σ partially during outputting. The second approach requires a lot of memory but the output procedure is simple. Algorithm 2 shows the first approach.

The function $R(i)$ in Algorithm 2 denotes the bigram of the right-hand side of the i-th rule α_i. For example, for $(\alpha_i \Rightarrow \beta\gamma) \in R$, $R(i) = (\beta, \gamma)$. In this algorithm, we identify the rule α_i by its subscript i while $\sigma[i]$ denotes the i-th non-terminal symbol on the right-hand side of σ.

5. Experiments

5.1 Compression Performance

We implemented our proposed algorithm, known as Re-Pair-VF, and compared it to the STVF algorithm (STVF) [8], the original Re-Pair algorithm [*3] (Re-Pair), BPEX [16], gzip, bzip2, and LZMA. We measured the compression ratios and the compression and decompression times. We used the default options

[*3] http://ihome.cuhk.edu.hk/~b126594/en/restore.html

Table 1 Text files used in our experiments.

Texts	Size (byte)	$\|\Sigma\|$	Contents
Dazai.utf.txt	7,268,933	141	Japanese texts (encoded by UTF-8)
DBLP2003.xml	90,510,236	97	XML data
GBHTG119.dna	87,173,787	4	DNA sequences
Reuters21578.txt	18,805,335	103	English texts

Table 2 Compression ratios as percentages.

	Re-Pair-VF	Re-Pair	STVF	BPEX	gzip	bzip2	LZMA
Dazai.utf.txt	25.86	21.90	33.74	32.14	33.41	22.93	23.06
DBLP2003.xml	13.67	11.04	22.08	19.11	17.30	11.26	11.62
GBHTG119.dna	28.01	23.84	24.07	28.12	28.23	26.00	23.36
Reuters21578.txt	27.96	23.40	37.21	33.60	36.98	25.80	23.87

Table 3 The maximum dictionary size (denoted by $\max(s)$) and the best size of dictionary (denoted by $\hat{s}$).

	$\max(s)$	$\hat{s}$
Dazai.utf.txt	247,599	123,812
DBLP2003.xml	1,560,135	780,037
GBHTG119.dna	1,928,611	964,183
Reuters21578.txt	694,245	347,098

Table 4 Compression times in seconds for each dataset.

	Re-Pair-VF	Re-Pair	STVF	BPEX	gzip	bzip2	LZMA
Dazai.utf.txt	3.536	7.372	1,240.854	22.953	0.752	0.820	7.048
DBLP2003.xml	41.339	109.287	1,609.241	145.601	2.528	14.925	54.279
GBHTG119.dna	46.959	152.454	1,708.855	84.489	17.513	11.561	160.690
Reuters21578.txt	10.881	25.002	1,395.139	54.919	1.268	2.416	20.637

Table 5 Decompression times in seconds for each dataset.

	Re-Pair-VF	Re-Pair	STVF	BPEX	gzip	bzip2	LZMA
Dazai.utf.txt	0.064	0.160	0.680	0.248	0.064	0.312	0.132
DBLP2003.xml	0.972	1.548	2.048	3.032	0.596	2.628	0.972
GBHTG119.dna	1.008	2.168	3.444	2.832	0.744	4.128	1.676
Reuters21578.txt	0.224	0.552	1.168	0.660	0.172	0.796	0.368

for gzip and bzip2. Re-Pair-VF and STVF are variable-to-fixed length encoding methods, whereas Re-Pair, gzip, and bzip2 are variable-to-variable length encoding methods. Our program was written in C++ and compiled using g++ version 4.6. We performed the experiments on a workstation equipped with an Intel Xeon (R) 3 GHz CPU with 12 GB RAM, which operated Ubuntu 12.04.

We used XML data, DNA data, English texts, and Japanese texts in our experiments (see **Table 1** for details). "Dazai.utf.txt" was the complete works of Osamu Dazai [*4], which was written in Japanese and encoded by UTF-8. "DBLP2003.xml" comprised all of the 2003 data from dblp20040213.xml [*5]. "GBHTG119.dna" was a collection of DNA sequences from GenBank [*6], from which we eliminated all of the metadata, spaces, and line feeds. "Reuters21578.txt" (distribution 1.0) [*7] was a sample collection of English texts.

Table 2 shows the compression ratios for each file and the compression method, which we measured as the (compressed file size)/(original file size). As shown in the table, Re-Pair-VF was better than STVF and gzip for natural language texts. In particular, Re-Pair-VF was approximately 1.3 times better than gzip, whereas it was 1.2 times worse than Re-Pair.

Table 3 shows the maximum size of the dictionary $\max(s)$ and the best value $\hat{s}$ defined in the previous section. We can observe that $\hat{s}$ becomes almost half of $\max(s)$ from the result.

Table 4 shows the compression times. The results show that Re-Pair-VF was two times faster than Re-Pair. This means that $\hat{s}$ can be selected with no increase in the time requirements. Moreover, Re-Pair took longer to encode the rules with complicated methods.

Table 5 shows the decompression times. Re-Pair-VF was faster than Re-Pair and STVF, and approximately three times faster than bzip2.

5.2 Pattern Matching Performance

We also implemented pattern matching algorithms for Re-Pair-VF according to the methods of Kida et al. in 2003 [9] to compare the pattern matching performance with compressed texts using STVF, BPEX, and gzip (see the Appendix for compressed pattern matching). We used UNIX zgrep for pattern matching on the text compressed by gzip. We omitted the original Re-Pair algorithm from this experiment, because it needs to decode the variable length codes and thus the compressed pattern matching of it is slower than that of Re-Pair-VF. We chose patterns with lengths of 5–50 characters in the text. We measured the pattern matching times for 50 patterns of each length and calculated the average.

Tables 6–9 lists the results for the matching throughput performance, which was measured as (the original text length)/(the average time for pattern matching). A higher value was better in

[*4] http://j-texts.com/
[*5] http://www.informatik.uni-trier.de/~ley/db/
[*6] http://www.ncbi.nlm.nih.gov/genbank/
[*7] http://www.daviddlewis.com/resources/testcollections/reuters21578/

Table 6 Matching throughput with DBLP2003.xml (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	202.315	106.427	1,506.524	159.630
10	182.975	106.297	1,508.410	160.875
15	200.308	106.147	1,508.409	163.289
20	199.501	105.899	1,508.410	154.789
25	198.872	105.811	1,508.409	153.303
30	194.906	105.603	1,508.409	151.162
35	196.587	105.318	1,508.410	133.802
40	195.231	105.014	1,508.410	133.130
45	193.363	104.385	1,508.409	131.382
50	191.370	103.897	1,508.410	136.392

Table 7 Matching throughput with dazai.utf.txt (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	312.544	12.649	908.560	111.676
10	302.853	12.616	908.560	113.191
15	302.853	12.603	908.560	113.257
20	302.853	12.562	908.560	113.476
25	302.853	12.555	908.560	114.536
30	259.589	12.520	908.560	113.564
35	259.588	12.487	908.560	112.974
40	227.140	12.416	908.562	111.972
45	201.902	12.351	908.558	114.228
50	181.712	12.250	908.560	114.155

Table 8 Matching throughput with GBHTG119.dna (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	159.430	55.352	1,083.897	86.110
10	149.633	55.268	1,089.604	86.706
15	148.720	55.234	1,088.566	86.533
20	153.030	55.181	1,089.604	86.762
25	156.903	55.111	1,089.604	86.614
30	162.581	54.989	1,089.604	86.875
35	165.938	54.947	1,089.604	86.646
40	169.492	54.834	1,089.604	86.460
45	174.745	54.664	1,087.529	86.829
50	177.953	54.524	1,089.604	86.795

Table 9 Matching throughput with reuters21578.txt (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	223.859	27.319	783.507	108.591
10	210.896	27.257	783.507	108.739
15	223.859	27.246	783.507	109.371
20	223.045	27.253	783.507	109.727
25	214.294	27.193	783.507	110.222
30	203.092	27.165	783.507	106.255
35	195.877	27.058	783.507	104.893
40	188.042	26.932	783.507	104.687
45	180.809	26.796	783.507	99.784
50	174.113	26.632	783.507	101.029

the tables. Each left-most column labeled *m* indicates the pattern length. The tables show that the pattern matching performance of Re-Pair-VF was the fastest except for BPEX. In particular, Re-Pair-VF was 1.1–2.8 times faster than zgrep.

6. Conclusions

In this study, we proposed a new VF coding method based on the Re-Pair algorithm, which we named as Re-Pair-VF. The experimental results demonstrated that our proposed coding method was superior to existing VF codes in terms of the compression ratio and compression time. The Re-Pair-VF algorithm uses fixed-length codewords but it delivered good compression performance, which was similar to bzip2. We also showed that pattern matching in a text compressed using the proposed coding method could be performed much faster than ordinary decompress-then-search approaches such as zgrep.

Our proposed coding method improved the compression speed significantly compared with existing VF codes, such as STVF, but it was still slower than gzip and bzip2. Thus, further improvements are required in future studies. We also intend to develop VF codes that use online grammar-based compression methods such as Sequitur [20].

Acknowledgments This work was partly supported by a Grant-in-Aid for Young Scientists (KAKENHI:23700002) of JSPS.

References

[1] Amir, A. and Benson, G.: Efficient two-dimensional compressed matching, *Proc. DCC*, pp.279–288 (1992).
[2] Brisaboa, N.R., Fariña, A., Navarro, G. and Paramá, J.: Dynamic lightweight text compression, *ACM Trans. Inf. Syst.*, Vol.28, No.3, article 10 (2010).
[3] Brisaboa, N.R., Fariña, A., López, J.-R., Navarro, G. and Lopez, E.R.: A new searchable variable-to-variable compressor, *Proc. DCC*, pp.199–208 (2010).
[4] Brisaboa, N.R., Fariña, A., Navarro, G. and Esteller, M.F.: (S, C)-dense coding: An optimized compression code for natural language text databases, *Proc. SPIRE*, pp.122–136 (2003).
[5] Brisaboa, N.R., Iglesias, E.L., Navarro, G. and Paramá, J.R.: An efficient compression code for text databases, *Proc. ECIR*, pp.468–481 (2003).
[6] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A. and Shelat, A.: The smallest grammar problem, *IEEE Trans. Inf. Theory*, Vol.51, No.7, pp.2554–2576 (2005).
[7] Gage, P.: A new algorithm for data compression, *The C Users Journal*, Vol.12, No.2, pp.23–38 (1994).
[8] Kida, T.: Suffix tree based VF-coding for compressed pattern matching, *Proc. DCC*, p.449 (2009).
[9] Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A. and Arikawa, S.: Collage system: A unifying framework for compressed pattern matching, *Theoretical Computer Science*, Vol.298, No.1, pp.253–272 (2003).
[10] Kida, T., Takeda, M., Shinohara, A., Miyazaki, M. and Arikawa, S.: Shift-And approach to pattern matching in LZW compressed text, *Proc. CPM*, pp.1–13 (1999).
[11] Kieffer, J.C. and Yang, E.-H.: Grammar-based codes: A new class of universal lossless source codes, *IEEE Trans. Inf. Theory*, Vol.46, No.3, pp.737–754 (2000).
[12] Kieffer, J.C., Yang, E.-H., Nelson, G. and Cosman, P.: Universal lossless compression via multilevel pattern matching, *IEEE Trans. Inf. Theory*, Vol.46, No.4, pp.1227–1245 (2000).
[13] Klein, S.T. and Ben-Nissan, M.K.: Using Fibonacci compression codes as alternatives to dense codes, *Proc. DCC*, pp.472–481 (2008).
[14] Klein, S.T. and Shapira, D.: Improved variable-to-fixed length codes, *Proc. SPIRE*, pp.39–50 (2008).
[15] Larsson, N.J. and Moffat, A.: Offline dictionary-based compression, *Proc. DCC*, pp.296–305 (1999).
[16] Maruyama, S., Tanaka, Y., Sakamoto, H. and Takeda, M.: Context-sensitive grammar transform: Compression and pattern matching, *Proc. SPIRE*, pp.27–38 (2008).
[17] Moffat, A. and Turpin, A.: On the implementation of minimum redundancy prefix codes, *IEEE Trans. Comm.*, Vol.45, No.10, pp.1200–1207 (1997).
[18] Navarro, G. and Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. CPM*, pp.14–36 (1999).
[19] Navarro, G. and Tarhio, J.: Boyer-Moore string matching over Ziv-Lempel compressed text, *Proc. CPM*, pp.166–180 (2000).
[20] Nevill-Manning, C., Witten, I. and Maulsby, D.: Compression by induction of hierarchical grammars, *Proc. DCC*, pp.244–253 (1994).
[21] Rautio, J., Tanninen, J. and Tarhio, J.: String matching with stopper encoding and code splitting, *Proc. CPM*, pp.42–52 (2002).
[22] Savari, S.A. and Gallager, R.G.: Generalized Tunstall codes for sources with memory, *IEEE Trans. Inf. Theory*, Vol.43, No.2, pp.658–668 (1997).
[23] Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A. and Arikawa, S.: A Boyer-Moore type algorithm for compressed pattern matching, *Proc. CPM*, pp.181–194 (2000).
[24] Tjalkens, T.J. and Willems, F.M.J.: Variable to fixed-length codes for Markov sources, *IEEE Trans. Inf. Theory*, Vol.IT-33, No.2, pp.246–

257 (1987).
[25] Tunstall, B.P.: Synthesis of noiseless compression codes, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA (1967).
[26] Turpin, A. and Moffat, A.: Housekeeping for prefix coding, *IEEE Trans. Comm.*, Vol.48, No.4, pp.622 –628 (2000).
[27] Uemura, T., Yoshida, S., Kida, T., Asai, T. and Okamoto, S.: Training parse trees for efficient VF coding, *Proc. SPIRE*, pp.179–184 (2010).
[28] Yamamoto, H. and Yokoo, H.: Average-sense optimality and competitive optimality for almost instantaneous VF codes, *IEEE Trans. Inf. Theory*, Vol.47, No.6, pp.2174–2184 (2001).
[29] Yoshida, S. and Kida, T.: Effective Variable-Length-to-Fixed-Length Coding via a Re-Pair Algorithm, *Proc. DCC*, p.532 (2013).
[30] Ziv, J.: Variable-to-fixed length codes are better than fixed-to-variable length codes for Markov sources, *IEEE Trans. Inf. Theory*, Vol.36, No.4, pp.861–863 (1990).

Appendix

A.1 Compressed Pattern Matching

Kida et al. [9] proposed a unified framework, known as the *Collage system*, for representing a dictionary compressed text and also presented an Aho-Corasick-type pattern matching algorithm on the framework. We can derive a pattern matching algorithm systematically using the Collage system for a text compressed with any form of dictionary compression if it is within the framework. Thus, Re-Pair-VF, BPEX, STVF, and Tunstall can be represented by the Collage system.

The Collage system is defined by a pair $\langle D, S \rangle$ where D is a sequence of definition tokens and S is the text represented by a sequence of tokens in D. Each token X_k in D is expressed as $expr_k$. Each expression $expr_k$ has one of the following forms:

(I) a for $a \in \Sigma \cup \{\varepsilon\}$,
(II) $X_i X_j$ for $i, j < k$,
(III) ${}^{[j]}X_i$ for $i < k$ and an integer j,
(IV) $X_i^{[j]}$ for $i < k$ and an integer j, and
(V) $(X_i)^j$ for $i < k$ and an integer j.

The forms (I)–(V) are primitive assignment, concatenation, prefix truncation, suffix truncation, and j times repetition, respectively. During dictionary-based compression, each codeword corresponds to a token. Therefore, we identify a codeword using its corresponding token below. The string represented by the token X is denoted by $X.u$. When the input text is $Y_1.u, Y_2.u, \ldots, Y_y.u$, we have $S = (Y_1, Y_2, \ldots, Y_y)$.

To perform pattern matching on compressed texts using a collage system, we simulate a deterministic finite automaton $M = (\Sigma, Q, q_0, F, \delta)$, which accepts the input patterns where Q, q_0, F, and δ are a set of states, the initial state, a set of final states, and a transition function, respectively. We need two functions to simulate the automaton: $Jump : Q \times F(D) \to Q$ and $Output : Q \times F(D) \to \wp(\mathbb{N})$, where $F(D)$ is a set of codewords in D and $\wp(\cdot)$ is the powerset of a set.

The function *Jump* simulates the state transition of the automaton. This function takes the state s and codeword X as the input, and returns the state where the state of the automaton moves from state s when the input text is $X.u$. The function *Jump* is defined as $Jump(s, X) = \delta(s, X.u)$.

The function *Output* determines the occurrences of patterns. This function takes the state s and codeword X as input, and returns a set of nonnegative integers i so the automaton reaches its final state when it takes the prefix of $X.u$ with length i from the state s as its input. The function *Output* is defined as $Output(s, X) = \{|v| : v \text{ is a non-empty prefix of } X.u \text{ such that } \delta(s, v) \in F\}$.

The outline of the algorithm used to construct the functions *Jump* and *Output* is as follows: (i) construct an automaton that accepts the pattern, (ii) perform the following for each state s of the automaton and each codeword X, (ii-a) set $Jump(s, X) := \delta(s, X.u)$, (ii-b) if there exists an integer i such that $\delta(s, X.u[1 : i]) \in F$, add $\{i\}$ to $Output(s, X)$. The algorithm for pattern matching is as follows: (i) set the current state s to 0, (ii) perform the following for each codeword X in the compressed text, (iii) if $Output(s, X)$ is not empty, report the pattern occurrences; (iv) set s to $Jump(s, X)$. Please refer to Ref. [9] for further details.

Next, we discuss the time and space complexity of the procedures used to construct the functions *Jump* and *Output*, which perform pattern matching on compressed texts. We present the following theorems, which are proved in Ref. [9], where D, $\|D\|$, $height(D)$, S, and m denotes the dictionary, the size of the dictionary, the height of the syntax tree for the dictionary, the compressed sequence, and the length of the pattern, respectively.

Theorem 1 (Theorem 1 from Ref. [9]) The function $Jump(j, X)$ can be achieved in $O(\|D\| \cdot height(D) + m^2)$ time using $O(\|D\| + m^2)$ space, so that it replies in $O(1)$ time. If D contains no truncations, the time complexity becomes $O(\|D\| + m^2)$.

Theorem 2 (Theorem 2 from Ref. [9]) The procedure used to enumerate the set $Output(j, X)$ can be achieved in $O(\|D\| \cdot height(D) + m^2)$ time using $O(\|D\| + m^2)$ space, so that it runs in $O(height(X) + \ell)$ time, where ℓ is the size of the set $Output(j, X)$. If D contains no truncations, it can be achieved in $O(\|D\| + m^2)$ time and space, so that it runs in $O(\ell)$ time.

Theorem 3 (Theorem 3 from Ref. [9]) The problem of compressed pattern matching can be solved in $O((\|D\| + |S|) \cdot height(D) + m^2 + r)$ time using $O(\|D\| + m^2)$ space. If D contains no truncation, it can be solved in $O(\|D\| + |S| + m^2 + r)$ time.

Re-Pair-VF does not contain any truncations or repetitions, because each node is represented as the concatenation of a node and a character. According to Theorem 3, pattern matching on Re-Pair-VF codes is achieved in $O(\|D\| + |S| + m^2 + r)$ time and $O(\|D\| + m^2)$ space.

Satoshi Yoshida received his B.S. degree in Information Science and M.S. degree in Information Science from Hokkaido University in 2009 and 2011, respectively. He is currently a doctoral student in Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University. His research interests include Text Algorithms, Data Structures and Data Compression. He is a member of IPSJ. He is currently a research fellow of the Japan Society for the Promotion of Science since April 2012.

Takuya Kida received his B.S. degree in Physics, M.S. and Dr. (Information Science) degrees all from Kyushu University in 1997, 1999, and 2001, respectively. He was a Full-time Lecturer of Kyushu University Library from October 2001 to March 2004. He is currently an Associate Professor of Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, since April 2004. His research interests include Text Algorithms and Data Structures, Information Retrieval, and Data Compression. He is a member of IEICE, IPSJ, and DBSJ.

(Editor in Charge: *Hiroshi Sakamoto*)