HOKKAIDO UNIVERSITY

Title	Efficient Variable-to-Fixed Length Coding Algorithms for Text Compression
Author(s)	吉田, 諭史
Degree Grantor	北海道大学
Degree Name	博士(情報科学)
Dissertation Number	甲第11292号
Issue Date	2014-03-25
DOI	https://doi.org/10.14943/doctoral.k11292
Doc URL	https://hdl.handle.net/2115/55455
Type	doctoral thesis
File Information	Satoshi_Yoshida.pdf

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

Efficient Variable-to-Fixed Length Coding Algorithms for Text Compression

（テキスト圧縮に対する効率よい可変長-固定長符号化アルゴリズム）

Satoshi Yoshida

February, 2014

Abstract

Data compression is a technique for reducing the storage space and the cost of transferring a large amount of data, using redundancy hidden in the data. We focus on lossless compression for text data, that is, text compression, in this thesis. To reuse a huge amount of data stored in secondary storage, I/O speeds are bottlenecks. Such a communication-speed problem can be relieved if we transfer only compressed data through the communication channel and furthermore can perform every necessary processes, such as string search, on the compressed data itself without decompression. Therefore, a new criterion "ease of processing the compressed data" is required in the field of data compression. Development of compression algorithms is currently in the mainstream of data compression field but many of them are not adequate for that criterion. The algorithms employing variable length codewords succeeded to achieve an extremely good compression ratio, but the boundaries between codewords are not obvious without a special processing. Such an "unclear boundary problem" prevents us from direct accessing to the compressed data.

On the contrary, *Variable-to-Fixed-length coding*, which is referred to as VF coding, is promising for our demand. VF coding is a coding scheme that segments an input text into a consecutive sequence of substrings (called *phrases*) and then assigns a fixed length codeword to each substring. Boundaries between codewords of VF coding are obvious because all of them have the same length. Therefore, we can realize "accessible data compression" by VF coding. Nevertheless, VF coding was not paid much attention

so far. The author does not know the reason why development of an efficient VF coding algorithm has not made in spite of the fact that the theoretical analysis guarantees the same attainable compression ratio as the non-VF coding algorithms.

In this study, the author improved the performance of VF coding algorithms. We developed high-level and well-balanced VF coding algorithms for the four criteria of compression ratio, compression speed, decompression speed, and processing on the compressed data. It is achieved by improving compression ratio, compression speed, and decompression speed of VF coding algorithms beyond the level of typical ones.

The organization of this thesis as follows. In Chapter 2, we focus on basic notion and definition. In Chapter 3, we discuss an improvement of *AIVF coding* proposed by Yamamoto and Yokoo in 2001. Conventional VF coding methods originated by *Tunstall coding* use tree structures called parse trees as dictionaries. Although AIVF coding achieves better compression ratio than Tunstall coding by using multiple parse trees, it is known that it requires large amount of time and space during compression. We propose an improved method by constructing an integrated parse tree used in AIVF coding and then simulate the encoding of AIVF coding on it. Moreover, we give theoretical analysis of upper and lower bounds of the number of nodes in the integrated parse tree and the number of nodes reduced by the integration. We experimentally show that our proposed method runs faster than AIVF coding on natural language texts and so on.

In Chapter 4, we discuss a method of brushing up a parse tree constructed by existent VF coding methods. We propose a method that repeatedly deletes useless nodes that are in the parse tree to add nodes that are expected to be useful but not in the parse tree with reading the input text. The method constructs a parse tree that achieves a fairly good compression ratio. We experimentally show that application of this method to a parse tree generated by *STVF coding*, which is proposed by Kida in 2009, yields better compression ratios than those of gzip.

In Chapter 5, we show how to realize a VF coding method by combining grammar-based compression and fixed length codeword. Grammar-based compression is a compression method that models the input text by a grammar that generates it to encode the grammar. We give a VF coding method that combines *Re-Pair algorithm* proposed by Larsson and Moffat in 2000 and fixed length coding. Re-Pair algorithm is a compression method based on a simple grammar, which achieves extremely good compression ratio. We introduce a numerical formula calculating the total amount of dictionary and compressed data in order to determine which rule in the grammar generated by Re-Pair algorithm should be in the dictionary. This method is beyond the framework of conventional VF coding methods with parse trees. It achieves better compression ratio than gzip by 20% or more and faster compression than STVF coding by a factor of 40 on natural language text. We also show that its decompression speed reaches to the highest level of existent compression methods.

In Chapter 6, we give practical techniques to apply VF coding methods to large texts. We propose two methods: (i) compressing large texts by block division and dictionary sharing and (ii) faster access to arbitrary position specified in the original text on compressed text. The former is a technique to reduce memory usage by dividing the input text into fixed length blocks and then compress each blocks. We improve the compression ratio by sharing a part of dictionaries for all blocks. The latter is a problem of identifying the position on compressed data corresponding to specified one on original text. To solve this, decompressing the compressed data from the beginning is generally required. We propose a faster method for this problem by having a bit sequence of n bits and its fully indexable dictionary where n denotes the length of the input text. We experimentally show that the proposed method works faster than FOLCA proposed by Maruyama *et al.* in 2013 by a factor of 10.

Finally, we conclude this thesis and discuss future works in Chapter 7. Through this study, the author succeeded to develop coding algorithms that are accord with

ease of processing on the compressed data, while keeping comparable performance in compression ratio and decompression speed with the state-of-the art non-VF coding algorithms.

Acknowledgements

First of all, I would like to express my appreciation to my supervisor Professor Takuya Kida. He directed my research and gave me plenty of advice. I would not accomplish this work without his generous support.

I would like to indicate my gratefulness to Professor Thomas Zeugmann, Professor Shin-ichi Minato, and Professor Hiroki Arimura. I thank them for their patient guidance and charitable comments. They spared much time to guide me and gave me great opportunities for my growth through Global Center Of Excellence The Formation of a Center for Next Generation Information Technology That Supports Knowledge Creation Program and Japan Science and Technology Agency Exploratory Research for Advanced Technology Minato Discrete Structure Manipulation System Project.

I would like to thank everyone in Information Knowledge Network laboratory, especially Ms. Yu Manabe. I could do nothing without her help. I enjoyed my life in the laboratory. I would live a very bored life in the laboratory without them.

I would like to voice my thankfulness to my father, my mother, and my sister. My parents let me enter the graduate school. I thank them for their encouragement.

Finally, this research was partially supported by JSPS (Japan Society for the Promotion of Science) Research Fellowships for Young Scientists.

Contents

Abstract **i**

Acknowledgements **v**

1 Introduction **1**

1.1 Related Studies 5

2 Preliminaries **9**

2.1 Terminology and Notation 9

2.2 Tunstall Coding 10

2.3 Almost Instantaneous VF Coding 11

2.4 STVF Coding 15

2.5 Compressed Pattern Matching for VF Coding 29

3 Efficient Algorithm for AIVF Coding **35**

3.1 Virtual Multiple AIVF Tree 36

3.2 Bound Analysis of VMA Tree 42

3.3 Experiments 44

3.4 Chapter Summary 46

4 Dictionary Training Algorithm for Efficient VF Coding **51**

4.1 Reconstruction Algorithm 51

4.2 Speeding-up by Sampling 54
4.3 Experimental Results 55
4.4 Chapter Summary 62

5 Efficient VF Coding Algorithm Using Re-Pair Algorithm 63
5.1 Re-Pair Algorithm 64
5.2 Re-Pair-VF 66
5.3 Experiments 71
5.4 Chapter Summary 77

6 Application of VF Coding to Large Texts 79
6.1 Efficient VF Coding by Block Dividing and Shared Dictionaries 79
6.2 Direct Access on Compressed Texts by VF Coding 87
6.3 Experiments 90
6.4 Chapter Summary 98

7 Concluding Remarks 103
7.1 Summary of the Results 103
7.2 Future Researches 105

Chapter 1

Introduction

Data compression is a technique for reducing the storage space and the cost of transferring a large amount of data, using redundancy hidden in the data. There are two categories for data compression: *lossless compression* and *lossy compression*. The former guarantees that the original data can be completely restored from the compressed data. While in the latter category, due to distortion, the original data can not be reconstructed precisely from the compressed data. We generally use lossless compression methods for string data, often referred to as *texts*, because we need to read them without error. We therefore focus on lossless compression in this thesis.

Performance of data compression methods was classically evaluated from their compression ratio, because to reduce the space was the most significant matter in an era when hard disk drives were extremely expensive. Compressing data also allows us to reduce transfer costs. A number of data compression algorithms are devised to gain better compression ratio [33, 34].

The significant criteria have changed into speeds for compression and decompression because hard disk drives with large capacity and high speed data transfer have been available in low price[1]. Therefore, compression methods that emphasize high speed

[1]From 1980s to 2010s, the cost of hard disk drives per unit capacity became 10 millionth[2]. In

processing rather than compression ratio have attracted attention. Particularly, gzip, which is an elder compression tool based on Lempel-Ziv method [46, 47], is still used due to its well-balancedness among the three criteria, which are compression ratio, compression speed, and decompression speed, in spite of its milder compression ratio compared to the state of the art compression methods.

Today, said to be the era of big data, a huge amount of data that were difficult to manage in past is available. Such data include contents of social networking services such as Twitter and Facebook, access logs of web servers, and sensor data generated by global positioning system receivers. Performing fast data analysis on such massive data is strongly required. Since such data are massive but individual data are not significant, they are usually discarded after performing event processing or saving summarized meta information at present. In this case, we have to determine what information is necessary in advance. In other words, we can not perform data analysis on past data, which is previously unexpected. To reuse past data, all the data must be preserved.

To reuse a huge amount of data stored in secondary storage, I/O speeds are bottlenecks. Such a communication-speed problem can be relieved if we transfer only compressed data through the communication channel and furthermore can perform every necessary processes, such as string search and access to any position, on the compressed data itself without decompression. From this viewpoint, pattern matching and data mining on compressed data have been gathering a great deal of attention since the late 1990s [1, 2, 12, 14, 16, 37].

Development of compression algorithm is currently in the mainstream of data compression field but many of them are not adequate for that criterion. The algorithms employing variable length codewords succeeded to achieve an extremely good compression ratio, but the boundaries between codewords are not obvious without a special

1980s, computers were connected to the network via telephone line with 56 kbps or leased line. In 2010s, high speed data transfer with 100 Mbps is available.

[2] `http://www.mkomo.com/cost-per-gigabyte`

processing. To treat such compressed data, extracting codewords with dictionary from the beginning of them is required because all the codewords do not have the same length.

On the contrary, *Variable-to-Fixed-length coding*, which is abbreviated as *VF coding*, is promising for our demand. VF coding is a coding scheme that segments an input text into a consecutive sequence of substrings, called *phrases*, and then assigns a fixed length codeword to each substring. Boundaries between codewords of VF coding are obvious because all of them have the same length. Therefore, we can realize "accessible data compression" by VF coding. However, they were seldom used in practice. Although it is theoretically known that VF coding achieves the same compression ratios as compression methods that employ variable length codes, a VF coding algorithm that achieves the compression ratio did not exist so far. For example, Shibata *et al.* [35] evaluated compression methods from the viewpoint of faster searching on the compressed data, and they rediscovered Byte Pair Encoding (BPE) [10], which is a kind of VF coding, but it has mild compression ratio about 50% on natural language text at most, while gzip usually achieves better than 40%.

The objective of this study is to improve performances of VF coding methods. We developed high-level and well-balanced VF coding algorithms for the four criteria of compression ratio, compression speed, decompression speed, and processing on the compressed data. It is achieved by improving compression ratio, compression speed, and decompression speed of VF coding beyond the level of typical ones. As mentioned above, gzip is a popular compression method for its well-balancedness. A goal of this study is therefore to design a VF coding algorithm that achieves good compression performance as gzip.

To improve the performance of VF coding is a difficult problem because they employ fixed length codewords. The compression ratio of a data compression method generally depends on the set of strings, called *dictionary*, used during compression. Therefore,

the problem of improving the compression ratio of it is the one of how to generate the optimal dictionary. However, this problem can not be solved in practical time because constructing the optimal dictionary is known to be an NP-Hard problem. Hence, the disputed point is how to construct a better dictionary with a greedy method.

The organization of this thesis as follows. In Chapter 2, we focus on basic notion and definition. In Chapter 3, we discuss an improvement of *Almost Instantaneous VF coding* (*AIVF coding*) [44] proposed by Yamamoto and Yokoo in 2001. Conventional VF coding methods originated by *Tunstall coding* [38] use tree structures called parse trees as dictionaries. Although AIVF coding achieves better compression ratio than Tunstall coding by using multiple parse trees, it is known that it requires large amount of time and space during compression. We propose an improved method by constructing an integrated parse tree used in AIVF coding and then simulate encoding of AIVF coding on it. Moreover, we give theoretical analysis of upper and lower bounds of the number of nodes in the integrated parse tree and the number of nodes reduced by the integration.

In Chapter 4, we discuss a method of brushing up a parse tree constructed by existent VF coding methods. We propose a method that repeatedly deletes useless nodes that are in the parse tree to add nodes that are expected to be useful but not in the parse tree with reading the input text. The method constructs a parse tree that achieves a fairly good compression ratio. We experimentally show that application of this method to a parse tree generated by *Suffix Tree-based VF coding* (*STVF coding*) [45] yields better compression ratios than those of gzip.

In Chapter 5, we show how to realize a VF coding method by combining grammar-based compression [17] and fixed length codeword. Grammar-based compression is a compression method that models the input text by a grammar that generates it to encode the grammar. We give a VF coding method that combines *Re-Pair algorithm* [21] proposed by Larsson and Moffat in 2000 and fixed length coding. Re-Pair algorithm is a compression method based on a simple grammar which achieves extremely good

compression ratio. We introduce a numerical formula calculating the total amount of dictionary and compressed data in order to determine which rule in the grammar generated by Re-Pair algorithm should be in the dictionary. This method is beyond the framework of conventional VF coding methods with parse trees. It achieves better compression ratio than gzip by 20% or more and faster compression than STVF coding by a factor of 40 on natural language text.

In Chapter 6, we give practical techniques to apply VF coding methods to large texts. We propose two methods: (i) compressing large texts by block division and dictionary sharing and (ii) faster access to arbitrary position specified in the original text on compressed text. The former is a technique to reduce memory usage by dividing the input text into fixed length blocks and then compress each blocks. We improve the compression ratio by sharing a part of dictionaries for all blocks. The latter is a problem of identifying the position on compressed data corresponding to specified one on original text. To solve this, decompressing the compressed data from the beginning is generally required. We propose a faster method for this problem by having a bit sequence of n bits and its fully indexable dictionary where n denotes the length of the input text. We experimentally show that the proposed method works faster than FOLCA [22] proposed by Maruyama *et al.* in 2013 by a factor of 10.

Finally, we conclude this thesis and discuss future works in Chapter 7.

1.1 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 [15, 27, 28]. 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 [32,36]. 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* [4–7]. Dense codes parse an input text using a morphological analysis tool before encoding it with byte-oriented codewords. Klein and Ben-Nissan [19] devised a variation of the Dense Code by using Fibonacci codes for text compression. Although Dense Codes work well for natural language texts that all words are separated by spaces such as English texts, they are not effective on texts that each word can not be easily extracted such as Japanese texts or DNA data.

For VF coding methods, Klein and Shapira [20] and Kida [13] independently presented a VF coding method based on a suffix tree (STVF coding[3]). A frequency-based-pruned suffix tree is used as a parse tree in the STVF coding. STVF coding is also suitable for searching because it uses a static dictionary and the codeword boundaries are obvious (see Section 2.5 for compressed pattern matching on VF coding). The compression ratio of STVF coding is superior to that of classical VF coding methods such as Tunstall coding, 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 [45].

Various practical algorithms have also been developed for grammar-based compres-

[3] The method of [20] is referred to as DynC in their paper, where the encoding algorithm is slightly different from that used by [13].

sion. Bisection [18] is a grammar-based compression algorithm where the grammar belongs to the class of a straight-line program. Compression algorithms have also been presented for restricted context-free grammars [17, 21, 29]. For example, Re-Pair [21] and Sequitur [29] are particularly useful because of their good compression ratios.

Maruyama *et al.* [23] presented an excellent compression method based on context-sensitive grammar, known as BPEX[4]. This method can be viewed as an extension of Byte Pair Encoding (BPE) [10], 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 from the middle of the compressed data because any codeword in BPEX-compressed data depends on the preceding codeword.

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

Chapter 2

Preliminaries

In this chapter, we introduce basic terms and notations. We also describe a brief sketch of VF coding.

2.1 Terminology and Notation

Let Σ be a finite alphabet and Σ^* be the set of all strings over Σ. The length of a string $x \in \Sigma^*$ is denoted by $|x|$. The string whose length is 0 is called the *empty string* and is denoted by ε. Therefore, we have $|\varepsilon| = 0$. The concatenation of two strings, x_1 and $x_2 \in \Sigma^*$, is denoted by $x_1 \cdot x_2$, and is also written simply as $x_1 x_2$, if no confusion occurs.

The occurrence probability of string $x \in \Sigma^*$ in a text S is denoted by $\Pr_S(x)$. We define $\Pr_S(a)$ as (the number of times that symbol a occurs in text S)/(the length of text S) for $a \in \Sigma$. We also define $\Pr_S(\varepsilon) := 1$ for convenience. Although $\Pr_S(x)$ depends on S, we write it simply as $\Pr(x)$ when the target text is obvious from the context or when we treat it as the statistical feature of a given information source.

A tree in which each node has at most k children is called a *k-ary tree*. A node that has some children is called an *internal node* or an *inner node*, and a node that has no

children is called a *leaf node* or a *leaf*. The node that has no parent, i.e., the top of a tree, is called the *root node* or the *root*. Furthermore, in a k-ary tree, a node that has exactly k children is called a *complete internal node*, and also an internal node that is not complete is called an *incomplete internal node*. A tree in which all internal nodes are complete is called a *complete k-ary tree*.

For a tree or a forest T, the set of all leaves, the set of all incomplete internal nodes, and the set of all complete nodes in T are denoted by $\mathcal{L}(T)$, $\mathcal{I}(T)$, and $\mathcal{C}(T)$, respectively. The union of $\mathcal{L}(T)$ and $\mathcal{I}(T)$ is denoted by $\mathcal{N}(T)$, and the size of set S by $\#S$. Then, for example, the number of leaves can be denoted by $\#\mathcal{L}(T)$. For a node n, the number of children of n is called the *degree* of n, which is denoted by $d(n)$.

2.2 Tunstall Coding

In this section, we discuss Tunstall coding. Conventional VF coding methods use parse trees as dictionary when they encode and decode the input texts. Such VF coding methods parses the input text into variable length strings, called phrases, using the parse tree, and assigns a fixed length codeword to each of them. Hereafter, a node in the parse tree is identified by its corresponding phrase.

Tunstall coding uses a complete $|\Sigma|$-ary tree as a parse tree T, called a *Tunstall tree*. Each edge in the tree is labeled with a symbol in Σ. Each node corresponds to a string over Σ, which is spelled out from the root to the node. Each codeword, which is a binary string of length $\ell := \lceil \lg \#\mathcal{L}(T) \rceil$, is assigned to each leaf in the Tunstall tree, namely, there is a one-to-one correspondence between a codeword and a leaf. A given text is parsed into a consecutive sequence of phrases by the tree, and each phrase is encoded with the corresponding binary codeword. The encoding algorithm is as Algorithm 2.1. Decoding for Tunstall coding is performed as follows: (i) read codewords one by one; (ii) find the node corresponding to the codeword; and (iii)

output the phrase corresponding to the node.

Next, we consider the algorithm that constructs the parse tree that maximizes the average phrase length. It is assumed that the information source is memoryless. Then, we have $\Pr(xa) = \Pr(x)\Pr(a)$ for $x \in \Sigma^+, a \in \Sigma$. The algorithm that constructs the parse tree with at most M codewords is as Algorithm 2.2. Lines 1 and 2 compute the number of inner nodes and the number of leaves in T, respectively. Please note that a k-ary tree with m inner nodes has $m(k-1)+1$ leaves. Therefore, a parse tree that has at most M codewords has not more than $\lfloor (M-1)/(|\Sigma|-1) \rfloor$ inner nodes. The optimality of the parse tree created here is proved in [41].

2.3 Almost Instantaneous VF Coding

We now give a brief survey of AIVF coding [44], which is based on Tunstall coding. In order to improve its compression ratio, Yamamoto and Yokoo employed two techniques to develop AIVF coding, one of which is to assign codewords to the incomplete internal nodes in the parse tree, and the other is to use multiple parse trees. Let $\Sigma := \{a_1, \ldots, a_{|\Sigma|}\}$ and assume that all symbols in Σ are sorted in descending order of their occurrence probabilities, for convenience of discussion. That is, $i < j$ implies $\Pr(a_i) \geq \Pr(a_j)$. It is also assumed that all the codes discussed below are binary codes.

2.3.1 Improvement by Assigning Codewords to Internal Nodes

In Tunstall tree, unused codewords of length ℓ exist if $\#\mathcal{L}(T) \neq 2^\ell$. This suggests that the average phrase length can be increased by assigning these unused codewords to some strings. If a leaf is added to a complete $|\Sigma|$-ary tree, an incomplete internal node is formed. That is, this also suggests that the average phrase length can be increased further if low-frequency leaves can be removed and the useful edges can be extended. Figure 2.1 is an example of the parse tree for this method, where $\Sigma = \{a, b, c\}$, the

Algorithm 2.1 Encoding algorithm for Tunstall coding.

Input: Parse tree T and an input text.

Output: An encoded text.

$n \leftarrow$ the root of T.
while not the end of the input text **do**
    $c \leftarrow$ the next symbol of the input text.
    $n \leftarrow$ the child of n labeled with c.
    if n is a leaf **then**
        Output a codeword assigned to n.
        $n \leftarrow$ the root of T.
    end if
end while

Algorithm 2.2 Constructing a Tunstall tree T with at most M codewords.

Input: The number of codewords M and occurrence probability $\Pr(a)$ of every character a.

Output: Tunstall tree T.

$m \leftarrow \lfloor (M-1)/(|\Sigma|-1) \rfloor$.
$M' \leftarrow m(|\Sigma|-1)+1$.
Create the root node of T.
$i \leftarrow |\Sigma|$.
while $i < M'$ **do**
    $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{L}(T)} \Pr(n)$.
    Create $|\Sigma|$ children of $\hat{n}$.
    $i \leftarrow i + |\Sigma| - 1$.
end while
return T.

occurrence probabilities are $0.6, 0.3, 0.1$, respectively, and the codeword length ℓ is equal to 3. All leaves and incomplete internal nodes have their own codewords, that is, for all $x \in \mathcal{N}(T)$, x is assigned a codeword.

2.3.2 Improvement by Using Multiple Parse Trees

In the method mentioned in the previous section, phrases are not statistically independent, even if the information source is memoryless. Parsing with a parse tree in which incomplete internal nodes have codewords causes contexts between phrases. Yamamoto and Yokoo also showed that the average phrase length can be increased by using a set of parse trees in order to catch the contexts. For example, assume that phrase aa is currently parsed and 001 is output, while encoding is performed using the parse tree shown in Figure 2.1. In this case, the next symbol is b or c, because the traverse had to be continued if the next symbol was a, and thus, phrase aaa should be parsed and 000 should be output. Therefore, when 001 is output, the nodes corresponding to the codes $000, 001, 010$, and 011 are unreachable in the next traverse. This suggests that the average phrase length can be increased by assigning these unreachable codewords to other strings. In this example, instead of using only one tree in Figure 2.1, we also use the tree in Figure 2.2. When the traverse fails at an internal node with a child labeled by a, we use the tree in Figure 2.2 for the next traverse. When we reach a leaf, we use the tree in Figure 2.1.

In the method proposed in [44], $|\Sigma| - 1$ parse trees T_i $(i = 0, \ldots, |\Sigma| - 2)$ are utilized. For each i, ith parse tree T_i has the root and its children are labeled by $a_{i+1}, \ldots, a_{|\Sigma|}$ $(i = 0, \ldots, |\Sigma| - 2)$ (recall that $\Pr(a_i) \geq \Pr(a_j)$ for $i < j$). A given text is encoded and decoded by switching the parse trees according to the context.

We now discuss how to construct the optimal parse tree for each T_i. Let M be the number of codewords. Then, our aim is to construct the optimal parse tree that maximizes the average phrase length $\sum_{x \in \mathcal{N}(T)} |x| \cdot \Pr(x)$ for a memoryless information

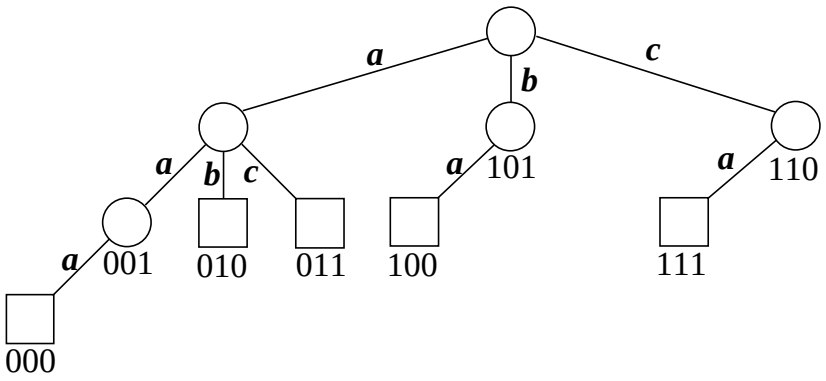

Figure 2.1: Parse tree for method in Section 2.3.1. A circle indicates an internal node and a square does a leaf.

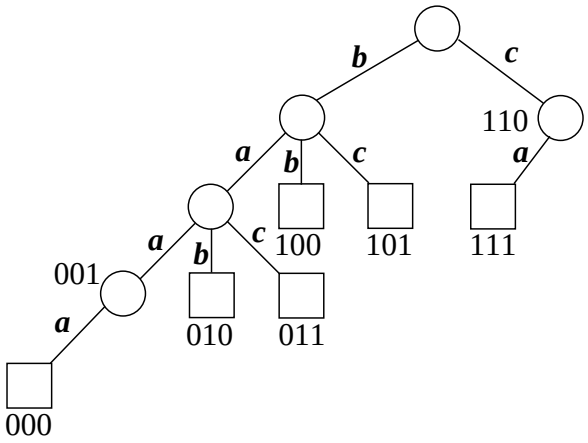

Figure 2.2: Another parse tree for AIVF coding.

source. The algorithm is shown in Algorithm 2.3. It should be noted that, when the $(|\Sigma|-1)$th child of an incomplete node is added, the average phrase length can be increased by adding the $|\Sigma|$th child without increasing the number of codewords, because a codeword is not assigned to a complete internal node. Recall that it is assumed that $\Pr(a_i) \geq \Pr(a_j)$ for $i < j$. In this case, a greedy algorithm yields the optimal solution.

The encoding algorithm with multiple parse trees is Algorithm 2.4. First, T_0 is selected as the current parse tree. Then, the symbols are read one by one, and the parse tree is traversed by the symbol. If the child labeled by the symbol does not exist, the codeword assigned to the node is output. Next, it must be determined which tree is used for the next traverse. Let n be the node that the preceding traverse finally reached, and let n have $d(n)$ children labeled by $a_1, a_2, \ldots, a_{d(n)}$. Then, the next symbol is larger than $a_{d(n)}$. Thus, the root of the next tree should have $|\Sigma| - d(n)$ children labeled by $a_{d(n)+1}, a_{d(n)+2}, \ldots, a_{|\Sigma|}$. The root of the tree T_i of the multiple parse trees has $|\Sigma| - i$ children labeled by $a_{i+1}, a_{i+2}, \ldots, a_{|\Sigma|}$. Hence, $T_{d(n)}$ is selected for the next parse tree and we jump to the root of $T_{d(n)}$. For example, let an input text be $S := aabaabaccab$, and consider encoding S with the parsing trees in Figures 2.1 and 2.2. Then, S is split into phrases, $aa \cdot baa \cdot bac \cdot ca \cdot b$. Therefore, the output binary sequence of length 15 bits is obtained as $001 \cdot 001 \cdot 011 \cdot 111 \cdot 101$. For the decoding, the same tree must be used as for the encoding.

2.4 STVF Coding

In this section, after explaining *suffix tree* [11,24,40,43], which is an index structure for strings, we illustrate STVF coding, which is a VF coding method based on suffix tree. Moreover, we give a method for improving its compression ratio using a technique in Section 2.3.1 here.

Algorithm 2.3 Constructing multiple parse trees.

Input: Alphabet Σ and occurrence probabilities of all characters $\Pr(a_1), \ldots, \Pr(a_{|\Sigma|})$.

Output: Parse trees $T_0, \ldots, T_{|\Sigma|-2}$.

for $i \leftarrow 0, 1, \ldots, |\Sigma| - 2$ **do**
  Create initial trees T^* which consists of the root node and its $|\Sigma| - i$ children.
  $m \leftarrow |\Sigma| - i$.
  $\hat{n} \leftarrow \operatorname{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n)$.
  while $|\Sigma| - d(\hat{n}) - 1 \leq M - m$ **do**
    $S_1 \leftarrow \Pr(\hat{n}) \sum_{j=d(\hat{n}+1)}^{|\Sigma|} \Pr(a_j)$.
    $S_2 \leftarrow 0$.
    for $|\Sigma| - d(\hat{n}) - 1$ times **do**
      $\tilde{n} \leftarrow \operatorname{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$.
      $S_2 \leftarrow S_2 + \Pr(\tilde{n})$.
    end for
    if $S_1 \geq S_2$ **then**
      Make $\hat{n}$ complete.
    else
      for $|\Sigma| - d(\hat{n}) - 1$ times **do**
        $\tilde{n} \leftarrow \operatorname{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$.
        Add the $(d(\tilde{n}) + 1)$th child of $\tilde{n}$.
      end for
    end if
    $m \leftarrow m + |\Sigma| - d(\hat{n}) - 1$.
    $\hat{n} \leftarrow \operatorname{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n)$.
  end while
  for $M - m$ times **do**
    $\tilde{n} \leftarrow \operatorname{argmax}_{n \in \mathcal{N}(T^*)} \Pr(n) \Pr(a_{d(n)+1})$.
    Add the $(d(\tilde{n}) + 1)$th child of $\tilde{n}$.
  end for
  $T_i \leftarrow T^*$.
end for
return $T_0, \ldots, T_{|\Sigma|-2}$.

Algorithm 2.4 Encoding with multiple parse trees.

Input: Parse trees $T_0, \ldots, T_{|\Sigma|-2}$ and text T.

Output: An encoded text.

```
1: T ← T_0.
2: n ← the root of T_0.
3: while not the end of the input text do
4:     c ← the next symbol of the input text.
5:     if there is no child of n in which we can traverse by symbol c then
6:         Output the codeword of n.
7:         T ← T_{d(n)}.
8:         n ← the root of T.
9:     else
10:        n ← the child of n labeled by c.
11:    end if
12: end while
```

2.4.1 Suffix Trees

The suffix tree of a given text T is a compacted trie which represents all the suffixes of T. 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. For a node v, we denote the label of the incoming edge of v by $label(v)$.

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[1]. Then, each leaf of $ST(T\$)$ corresponds one-to-one with each suffix of T, where \$ is a special symbol not in Σ.

For example, the suffix tree of a string BABCABABBABCBAC\$ is shown in Figure 2.3. For a node v in $ST(T\$)$ and a symbol $c \in \Sigma$, the function $child(v, c)$ and $f(v)$ return the child of v whose label of the incoming edge starts with c and the number of occurrences of $str(v)$ in T, the frequency of a node v, respectively. Since the frequency of node v is equal to the number of leaves in the subtree rooted at v, the computation of the frequencies of all nodes is done in a post-order traversal. Note that the suffix tree for T is constructed in linear time in the length of T.

2.4.2 VF Coding by Pruned Suffix Tree

In this section, we introduce a VF coding method that uses a pruned suffix tree for a parse tree, which is named as STVF coding and firstly (and partially) presented in [13].

[1]The representing string of the root node is the empty string, that is, $str(root) = \varepsilon$.

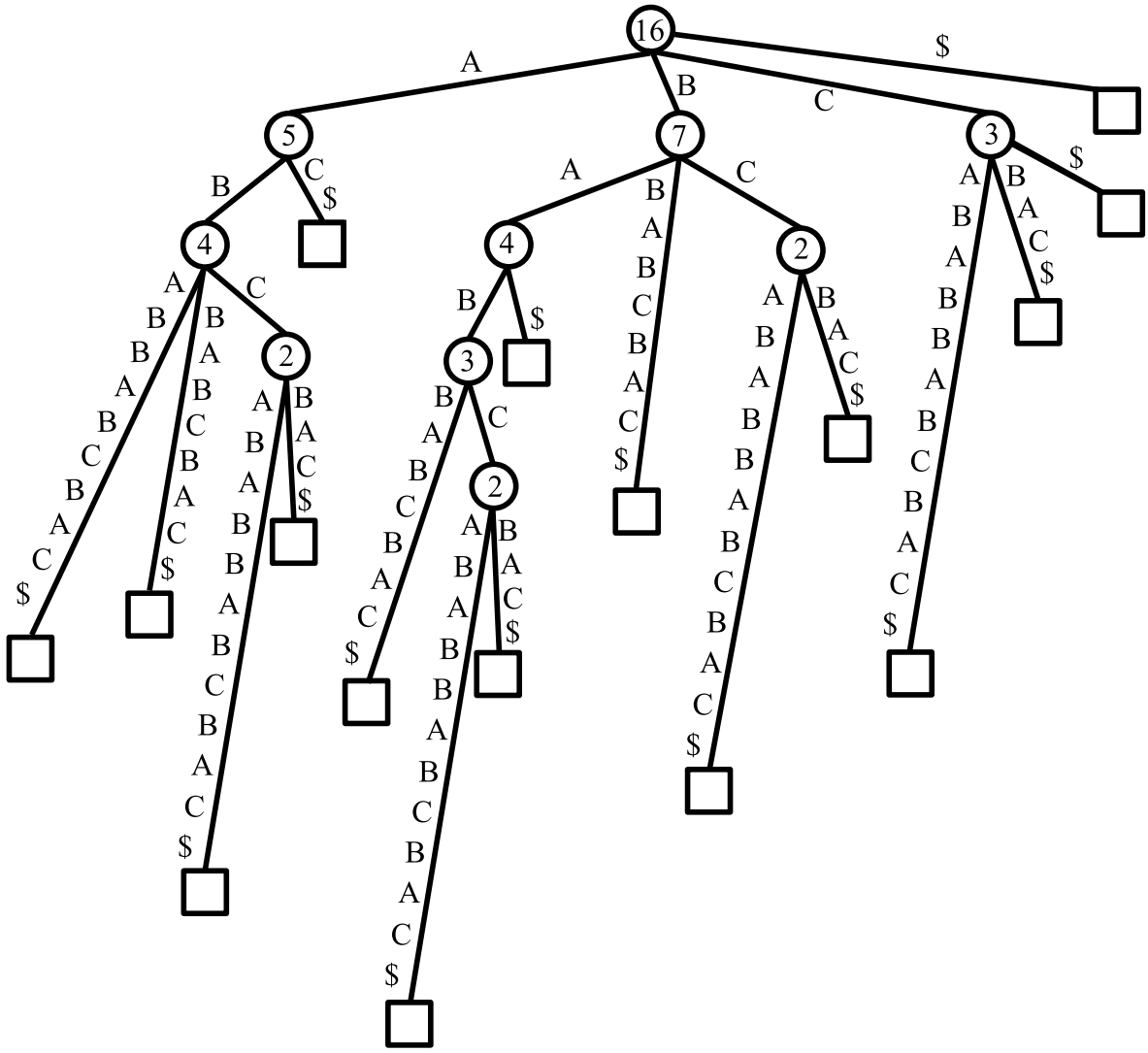

Figure 2.3: Suffix tree for string BABCABABBABCBAC$. The squares represent leaves. The circles represent internal nodes and the numbers in the circles are their frequencies.

Tunstall coding does not achieve a good compression ratio for an information source with memory because it assumes that the information source is memoryless. As stated in the previous section, a suffix tree stores all substrings of the given text; moreover the frequency of any substring can be easily obtained. This suggests that the suffix tree can be a good base of the parse tree for the given text. For a given text T, the deepest leaf, which is the leaf v such that $str(v)$ is the longest among all leaves, represents T itself. Therefore the whole $ST(T)$ can not be used as a parse tree. The idea of our new VF coding method is to prune deeper nodes in $ST(T)$ and make it a compact parse tree.

We denote by $ST_L(T)$ a pruned suffix tree such that the number of leaves equal to L obtained by pruning nodes corresponding to infrequent substrings. An internal node u in the parse tree is said to be complete if the parse tree contains all the children of u in $ST(T)$, otherwise u is said to be incomplete here. Note that a pruned suffix tree includes all nodes whose depth is 1 that are also included in the original $ST(T)$ so that it includes any symbols which occur in T. Now we consider to encode T by codewords of length ℓ. As the same as the Tunstall coding, the formula $L \leq 2^\ell$ must be satisfied. The procedure to parse and encode T with $ST_L(T)$ is also the same way as Tunstall coding.

The simplest strategy of pruning is to search $ST(T)$ by breadth-first-search from the root, and select the shallowest nodes till the number of leaves in a pruned suffix tree is up to L. A more sophisticated way is to select the nodes so that the frequencies of the leaves in $ST_L(T)$ become nearly uniform. Namely, select the nodes in the descending order of their frequencies from the root. Algorithm 2.5 is the parse tree construction algorithm of the STVF coding, and Figure 2.4 shows the parse tree $ST_8(T)$ 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 (Lines 4–14), the most frequent node v among the leaves in the temporal parse

tree $\mathcal{T}_i$ is selected, and then all the children of v in $ST(T)$ are added to $\mathcal{T}_i$. If the child is a leaf in $ST(T)$, the algorithm removes its label string of the incoming edge except for the first character. After the above pruning steps, the algorithm assigns codewords to all the leaves in a left-to-right manner. The first four iterations of the constructing process for the running example is shown in Figure 2.5. This construction strategy is similar to that of Tunstall coding.

For the parse tree $ST_{L'}(T)(L' \leq L)$ obtained by the algorithm, we have the following lemma.

Lemma 2.1. *For a given text T, we can uniquely parse T by using the pruned suffix tree $ST_{L'}(T)$.*

Proof. Let D be a set of strings which is entered into the pruned suffix tree $ST_{L'}(T)$, and call D as a dictionary. From the pruning procedure each leaf in $ST_{L'}(T)$ corresponds one-to-one to each string entered in D. Therefore, all the strings in D satisfy the prefix condition since only leaves are assigned the codewords, that is, for any string $s \in D$, there exists no string $t \in D$ such that $t \neq s$ and s is a prefix of t. Hence, we can uniquely parse the input text T. □

Once the parse tree is constructed, the encoding and decoding procedures are simple: they are shown in Algorithms 2.6 and 2.7, respectively. For the running example in Figure 2.4, the text is parsed into seven substrings as BA · BC · ABA · BB · ABC · BA · C, and encoded to 100 · 110 · 000 · 101 · 010 · 100 · 111. The parse tree must be stored together with the sequence of encoded phrases. We divide the parse tree into two components: the tree structure and the labels on it. The tree structure is encoded by balanced parentheses [26]. Thus the encoded size for the tree of M nodes is $2M$ bits. For the labels, we store them by a simple way: enumerate pairs of the label length and the label string and then attach to the encoded tree structure. Assuming that each label length is smaller than 256, which can be represented by one byte, the set of labels can

Algorithm 2.5 Algorithm of constructing a parse tree of STVF coding.

Input: Text T and codeword length ℓ.

Output: A parse tree.

Construct the suffix tree $ST(T)$ of T.
Construct the initial tree $\mathcal{T}$ which only contains the root of $ST(T)$.
$U \leftarrow \{root\}$. ▷ Set of nodes that will be assigned codewords.
while $|U| < 2^\ell$ **do**
    $v \leftarrow \text{argmax}_{v \in U} f(v)$.
    $U \leftarrow U \setminus \{v\}$.
    for all child w of v **do**
        $U \leftarrow U \cup \{w\}$.
        if w is a leaf of $ST(T)$ **then**
            Remove $lavel(w)$ except for the first character of it.
        end if
        Add w to $\mathcal{T}$.
    end for
end while
Assign codewords to the elements in U.
return $\mathcal{T}$ as $ST_{|U|}(T)$.

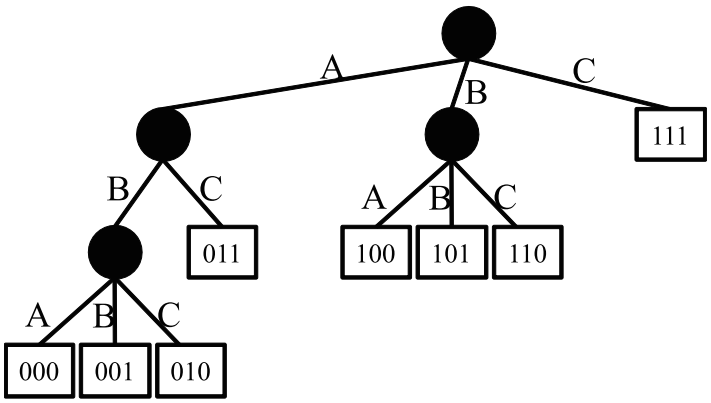

Figure 2.4: Parse tree of the STVF coding for string BABCABABBABCBAC. The squares and the circles indicate leaves and internal nodes, respectively. The numbers in squares are assigned codewords.

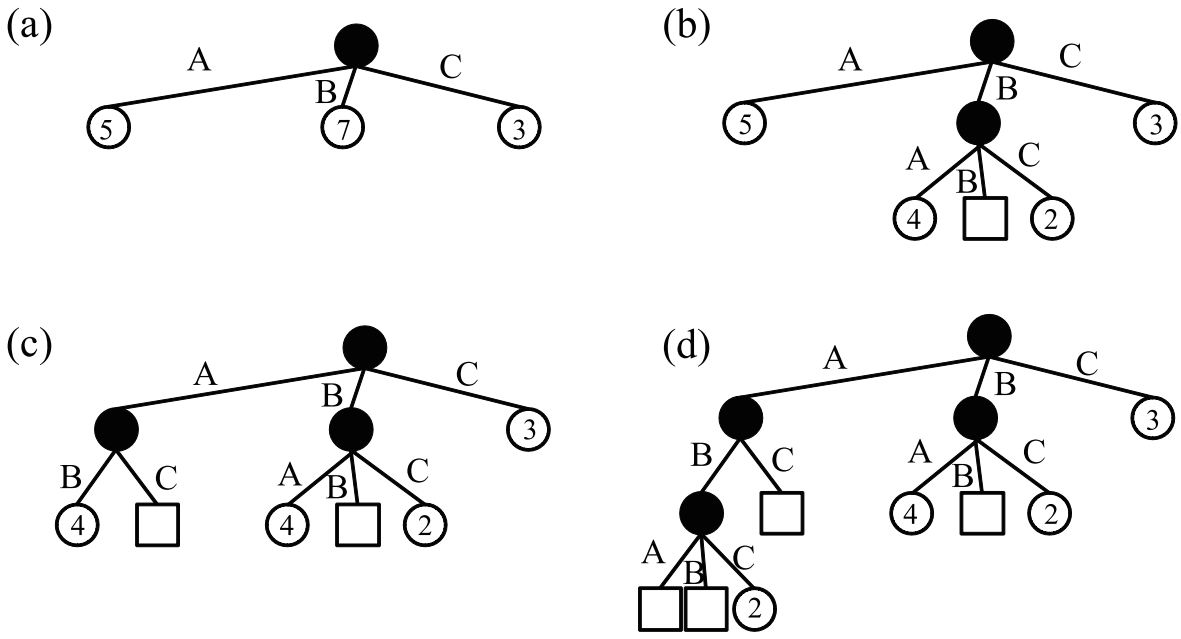

Figure 2.5: The first four iterations of the construction process of the parse tree. The black circles indicate internal nodes. Only leaves are assigned codewords.

be stored by $\sum_{x \in \mathcal{X}}(|x| + 1)$ bytes, where $\mathcal{X}$ is the set of labels.

The following lemma and theorem suggest the performance of STVF coding.

Lemma 2.2. *The parse tree constructed by Algorithm 2.5 is equivalent to the Tunstall tree for a sufficiently long string of arbitrary memoryless information sources.*

Proof. For a node p in the suffix tree and a character c, $f(p) \cdot Pr(c) = f(p \cdot c)$ holds with a sufficiently long string because we assume that the information source is memoryless. Therefore, we obtain the occurrence probability of the representing string of each node by dividing the frequency of each node by that of the root node. Both of the parse tree construction algorithms of STVF coding and Tunstall coding select the leaf that has the maximum probability to add all its children. Therefore, both of the algorithms select the same node. Hence, the STVF tree and the Tunstall tree are equivalent for arbitrary memoryless information sources. □

Theorem 2.1. *The number of codewords output by STVF coding is the same as the one output by Tunstall coding for arbitrary memoryless information source.*

Proof. The parse tree of STVF coding is equivalent to the Tunstall tree for memoryless information sources from Lemma 2.2. Therefore, the number of codewords output by STVF coding and the one output by Tunstall coding are the same for arbitrary memoryless information sources. □

2.4.3 Improving the Compression Ratio by Almost Instantaneous Coding

Next, we present an improved version of STVF coding stated in the above. In STVF coding, unused codewords of length ℓ exist if there does not exist an integer m satisfying $m(|\Sigma| - 1) + 1 = 2^{\ell}$. This suggests that we can encode the input text with fewer codewords by assigning such unused codewords to some strings. If we add a leaf to a

Algorithm 2.6 Encoding algorithm for STVF coding.

Input: Text T and parse tree $\mathcal{T}$.

Output: An encoded text.

```
1: i ← 0.
2: while i < |T| do
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
```

Algorithm 2.7 Decoding algorithm for the STVF coding.

Input: Parse tree $\mathcal{T}$ and sequence of codewords C.

Output: The original text.

```
1: for all i ∈ {0, ..., |C| − 1} do
2:     v ← the node such that code(v) = C[i].
3:     Output str(v).
4: end for
```

complete $|\Sigma|$-ary tree, an incomplete internal node is made. That is, this also suggests that we can acquire much better compression ratios if we remove low-frequency leaves and extend useful edges.

We introduce the algorithm for constructing a parse tree as in Algorithm 2.8. The basic idea of the algorithm is to choose the most frequent node from the suffix tree, which 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 coding algorithm that extends all the children of the chosen node at once. Figure 2.6 is an example of the parse tree constructed by the algorithm of Algorithm 2.8 for $T :=$ BABCABABBABCBAC. Now we explain the move of the algorithm. For a given text T, we first construct the suffix tree $ST(T)$ and remove the labels of the leaves in $ST(T)$ except for the first characters of them. Let U be the set of nodes which will be assigned codewords and V be the set of candidate nodes for phrases which are in $ST(T)$ but not in the parse tree. Note that each node in V is a child of a node in U. Initially, U is the empty set and V is the children of the root of $ST(T)$. Next, to ensure the algorithm encodes the text correctly, we add all the children of the root to U. Then, we repeat the following procedure while $|U| < 2^{\ell}$: we select the node v whose frequency is maximal in V. Then, we add it to U and delete it from V. If there remains just one node $w \in V$ that is a sibling of v, we add w to U and delete its parent from U. It is not necessary to assign a codeword to a complete node because the traversals in the encoding process never fail at any complete nodes. The node p is now complete and thus it will not be assigned a codeword. Finally, we assign unique codewords to the elements in U in a left-to-right manner.

Figure 2.7 shows the construction process of the parse tree for the running example by the algorithm. The input string is parsed into five substrings by using the parse tree in Figure 2.6, as BABC$\cdot$AB$\cdot$AB$\cdot$BABC$\cdot$BAC, and encoded to 101$\cdot$000$\cdot$000$\cdot$101$\cdot$110. In this case, the encoded length is shorter than that of the STVF coding in the previous

Algorithm 2.8 Improved construction algorithm for parse trees.

Input: Text T and codeword length ℓ.

Output: A parse tree.

1: Construct the suffix tree $ST(T)$ of T.
2: Construct the parse tree $\mathcal{T}$ which only contains the root of $ST(T)$.
3: $U \leftarrow \emptyset; V \leftarrow \{v \mid v \text{ is a child of the root of } ST(T)\}$.
4: **for all** Child v of the root of $ST(T)$ **do**
5: Add v to $\mathcal{T}$.
6: **if** v corresponds to a leaf in $ST(T)$ **then**
7: Remove $label(v)$ except for the first character of it.
8: **end if**
9: $U \leftarrow U \cup \{v\}$.
10: $V \leftarrow (V \setminus \{v\}) \cup \{w \mid w \text{ is a child of } v\}$.
11: **end for**
12: **while** $|U| < 2^\ell$ **do**
13: $v \leftarrow \text{argmax}_{v \in V} f(v)$.
14: Add v to $\mathcal{T}$.
15: $U \leftarrow U \cup \{v\}$.
16: $V \leftarrow (V \setminus \{v\}) \cup \{w \mid w \text{ is a child of } v\}$.
17: $p \leftarrow v$'s parent.
18: **if** $\#\{w \in V \mid w \text{ is a child of } p\} = 1$ **then**
19: $w \leftarrow p$'s just one child remaining in V.
20: $U \leftarrow (U \setminus \{p\}) \cup \{w\}$.
21: $V \leftarrow (V \setminus \{w\}) \cup \{x \mid x \text{ is a child of } w\}$.
22: **end if**
23: **end while**
24: Assign codewords to the elements in U.
25: **return** $\mathcal{T}$.

section.

We discuss the time and space complexities of Algorithm 2.8. Constructing the suffix tree $ST(T)$ needs $O(|T|)$ time and space. It is obvious that Line 2 takes $O(1)$ time. Since we can manage both set U and tree $\mathcal{T}$ just by marking nodes in $ST(T)$, adding or deleting an element for them is done in $O(1)$ time. To process Line 13 efficiently, we assume that the set V is realized by a priority queue based on a max-heap. That is, we need $O(\log |V|)$ time for adding or deleting an element for V, while answering the maximum element among V is done in $O(1)$ time. Then, Line 3 needs $O(|\Sigma|)$ time. For the loop in Lines 4–11, the number of iterations is $O(|\Sigma|)$. Thus, the time complexity of the loop is $O(|\Sigma| \log |\Sigma|)$ since the size of V can increase to $O(|\Sigma|^2)$. For the while loop in Lines 12–23, the number of iterations is restricted to the size of U, but the size of V is a dominant factor for the time complexity. We can calculate in $O(1)$ time for each line within the loop except for Lines 16 and 21[2]. The number of nodes added to V is $|T|$ at most, and the number of nodes deleted from V too. Thus, Lines 16 and 21 take $O(|T| \log |T|)$ time totally. Finally, Line 24 takes $O(|T|)$ time. Therefore, the total time complexity of the algorithm is $O(|\Sigma| \log |\Sigma| + |T| \log |T|)$. The complexity will be $O(|T| \log |T|)$ when $|\Sigma| \leq |T|$. For the space consumption, we need only $O(|T|)$ space since both U and $\mathcal{T}$ can be managed by adding $O(1)$ size information on each node of $ST(T)$, in addition to the priority queue whose maximal size is restricted to $|V|$, namely, $O(|T|)$.

Next, we show the encoding and the decoding algorithms. We need to modify the encoding algorithm because codewords can be assigned to internal nodes. It is shown in Algorithm 2.9. 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

[2] For Line 19, we need an auxiliary data structure on each node to do so. For example, it is realized by a doubly linked list between siblings.

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 call it the almost instantaneous encoding. The algorithms of decoding and storing the parse tree are common to the STVF coding algorithm except 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 k nodes are encoded to $3k$ bits.

The following lemma is important for the correctness of the encoding algorithm using the parse trees constructed by the algorithm in Algorithm 2.8.

Lemma 2.3. *Let T be a given text and $\mathcal{T}$ be the parse tree of T constructed by the Algorithm 2.8. For any suffix s of T, there exist at least one node in $\mathcal{T}$ which represents a nonempty prefix of s, and there exists one node which represents the longest prefix of s in $\mathcal{T}$ and which is also assigned a codeword.*

Proof. The former is clear because all the children of the root are contained in $\mathcal{T}$. We next prove the latter by a 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 $str(v)$. □

2.5 Compressed Pattern Matching for VF Coding

Kida *et al.* [14] 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, all VF coding methods treated

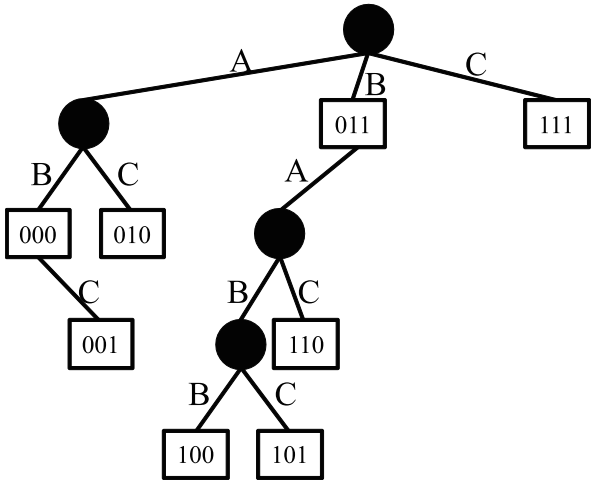

Figure 2.6: Parse tree of the method in Section 2.4.3 for string BABCABABBABCBAC. The squares represent the nodes assigned codewords, corresponding to the numbers in them. The circles represent the complete internal nodes.

Algorithm 2.9 Modified encoding algorithm.

Input: Text T and parse tree $\mathcal{T}$.

Output: An encoded text.

1: $i \leftarrow 0$.
2: **while** $i < |T|$ **do**.
3: $v \leftarrow root$.
4: **while** $str(v) \cdot T[i]$ is represented by $\mathcal{T}$ **do**
5: $v \leftarrow$ the node that represents $str(v) \cdot T[i]$.
6: $i \leftarrow i + 1$.
7: **end while**
8: **Output** the codeword assigned to v.
9: **end while**

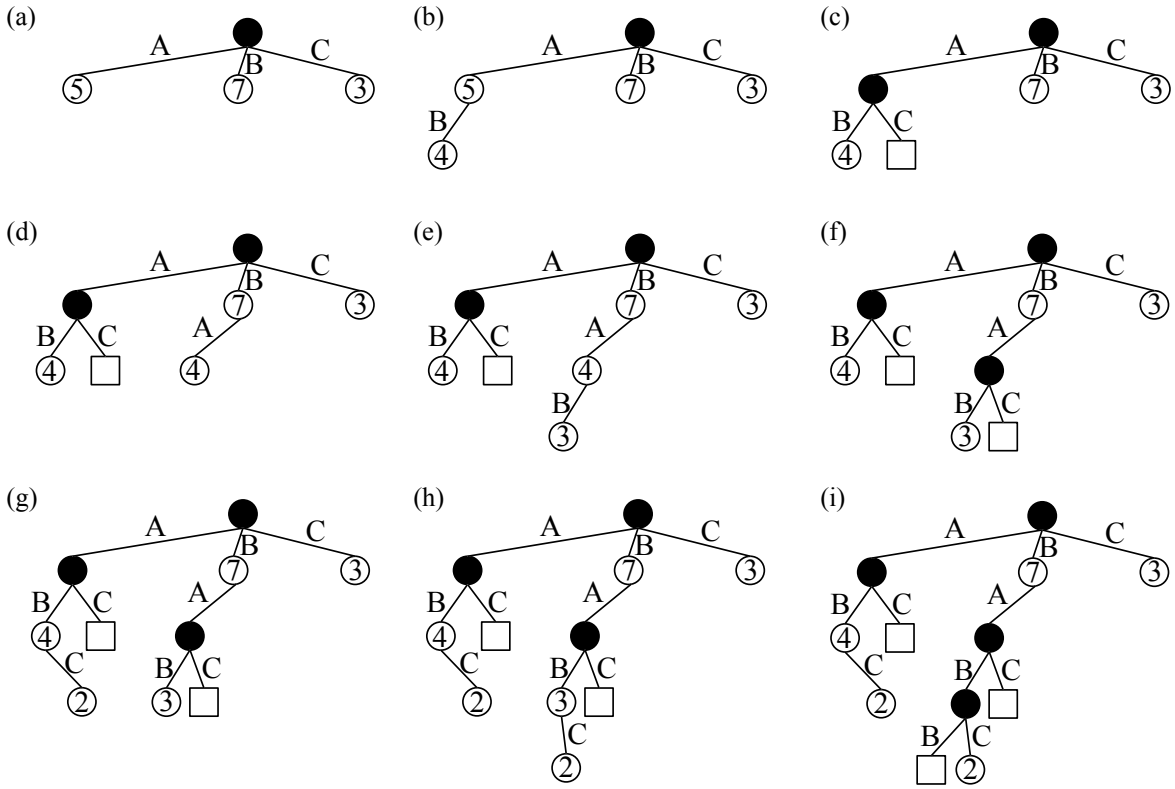

Figure 2.7: Iterations of constructing a parse tree with Algorithm 2.8. The black circles represent complete internal nodes, which are not assigned codewords.

in this thesis 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 $(\Sigma, Q, q_0, F, \delta)$, 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 [14] 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 [14], where D, $\|D\|$, $height(D)$, S, m, and r denotes the dictionary, the size of the dictionary, the height of the syntax tree for the dictionary, the compressed sequence, the length of the pattern, and the number of pattern occurrences, respectively.

Theorem 2.2 (Theorem 1 from [14]). *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 2.3 (Theorem 2 from [14]). *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 2.4 (Theorem 3 from [14]). *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.*

All VF coding methods treated in this thesis do not contain any truncations or repetitions, because each node is represented as the concatenation of a node and a character. According to Theorem 2.4, pattern matching on them is achieved in $O(\|D\|+|S|+m^2+r)$ time and $O(\|D\|+m^2)$ space.

Chapter 3

Efficient Algorithm for AIVF Coding

In this chapter, we propose an efficient algorithm for encoding and decoding of AIVF coding, which integrates the multiple parse trees into a compact single tree and simulates the encoding and the decoding procedure of the original AIVF coding. Our idea originated in the observation that many nodes in the multiple parse trees of an AIVF coding are common, and thus, they can be multiplexed (see Figure 3.1). We refer to the integrated parse tree as the *Virtual Multiple AIVF parse tree* (*VMA tree* for short). We prove that the upper and lower bounds of the number of nodes in the VMA tree are $M|\Sigma| - |\Sigma|^2/2 + |\Sigma|/(|\Sigma| - 1)$ and $M\ln(|\Sigma| + 1) - M/2 + 1$, respectively, where M denotes the number of codewords. We also prove that the upper and lower bounds of the number of nodes reduced from the original multiple parse trees are $[|\Sigma|M(|\Sigma| - 1/2) + (|\Sigma| - 6)(|\Sigma| + 1)/2 - |\Sigma|M\ln(|\Sigma| + 1)]/(|\Sigma| - 1)$ and $|\Sigma|^2/2 + 5|\Sigma|/2 - 7$, respectively. We show that in fact our technique allows much faster encoding than does the original AIVF coding for natural language texts.

3.1 Virtual Multiple AIVF Tree

In this section, we explain our idea and present an efficient algorithm for AIVF coding. In the parse trees of AIVF coding, it can be observed that many nodes in T_i are identical to those in T_{i+1}. More precisely, T_{i+1} completely covers the nodes in T_i, except for the leftmost[1] subtree under the node corresponding to a_{i+1}. First, we explain this relationship. Let $S_j^{(i)}$ be the subtree of T_i that consists of all the nodes under the direct child of the root corresponding to a_j. Then, we have the following theorem.

Theorem 3.1. *Subtree $S_{i+j}^{(i+1)}$ completely covers $S_{i+j}^{(i)}$ for any integers i $(0 \leq i \leq |\Sigma|-3)$ and j $(2 \leq j \leq |\Sigma| - i)$.*

Proof. We prove the theorem by contradiction. For an integer i $(0 \leq i \leq |\Sigma|-2)$, let T_i^{∞} be the tree of infinite depth in which the root has children according to $a_{i+1}, \ldots, a_{|\Sigma|}$, and all the other internal nodes have just $|\Sigma|$ children. It should be noted that the root of the multiple parse tree T_i has children according to a_j $(j = i+1, \ldots, |\Sigma|)$. That is, T_i includes $S_j^{(i)}$ $(j = i+1, \ldots, |\Sigma|)$. It should be also noted that each T_i is optimal in the sense that it maximizes the average phrase length. That is, for any i $(0 \leq i \leq |\Sigma| - 3)$ and j $(2 \leq j \leq |\Sigma| - i)$, the average phrase length cannot be further increased by exchanging any node in $S_{i+j}^{(i)}$ for a node included in T_i^{∞} but not in T_i. Here, we assume that $S_{i+j}^{(i+1)}$ does not cover $S_{i+j}^{(i)}$ completely. Then, there exists a node that is in $S_{i+j}^{(i)}$ but not in $S_{i+j}^{(i+1)}$. Let n be the node. Since T_i is a parse tree that maximizes the average phrase length, it can be increased by exchanging a node in $S_{i+j}^{(i+1)}$ but not in $S_{i+j}^{(i)}$ for n. However, this contradicts that T_{i+1} maximizes the average phrase length. Therefore, $S_{i+j}^{(i+1)}$ completely covers $S_{i+j}^{(i)}$. □

The set of trees can be multiplexed and simply integrated into a single tree according to the above theorem. To simulate the encoding and the decoding of the original AIVF

[1]Each edge is arranged in descending order of the occurrence probability of its label character in left to right manner.

coding by using the VMA tree, it is necessary to indicate which parse tree is currently being traversed during processing. Thus, each node in the VMA tree is marked to indicate to which trees the node belongs. Since a node can belong to several parse trees, the least i is saved such that n belongs to T_i for each node. Denoting this mark by $\mathrm{Tn}(n)$, we have that $\mathrm{Tn}(n) := \min_i\{i \mid 0 \leq i \leq |\Sigma| - 2, n \text{ belongs to } T_i.\}$. For example, by integrating the two parse trees shown in Figures 2.1 and 2.2, the parse tree shown in Figure 3.2 is obtained.

To encode with a VMA tree, the previous encoding algorithm must be modified, because even if there is a child in the VMA tree for the next traverse, the traverse fails when there is no child in T_i. Therefore, $\mathrm{Tn}(n)$ and the number i of the currently traversing tree T_i are compared. If i is less than $\mathrm{Tn}(n)$, there is not a proper node in T_i, and we return to the root. The encoding algorithm is shown in Algorithm 3.1, where the codeword assigned to n in T_i is denoted by $w_i(n)$.

The algorithm for constructing a VMA tree is shown in Algorithm 3.2. Let S_i denote the subtree that consists of all nodes under the root node corresponding to a_i. We define $\#\mathcal{N}(S_0) := M$ for convenience. We define the function $\mathrm{cod}(a_j) := j$, and denote by $\mathrm{first}(n)$ the first symbol of the label sequence on the path from the root to n. That is, if $\mathrm{first}(n) = a_j$, $\mathrm{cod}(\mathrm{first}(n)) = j$. The number of nodes having the codeword is denoted by m in Algorithm 3.2.

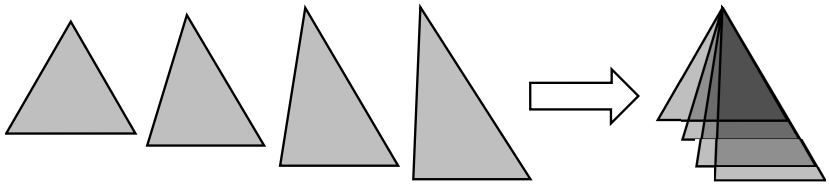

Figure 3.1: Multiplexing the parse trees of AIVF coding into a single tree.

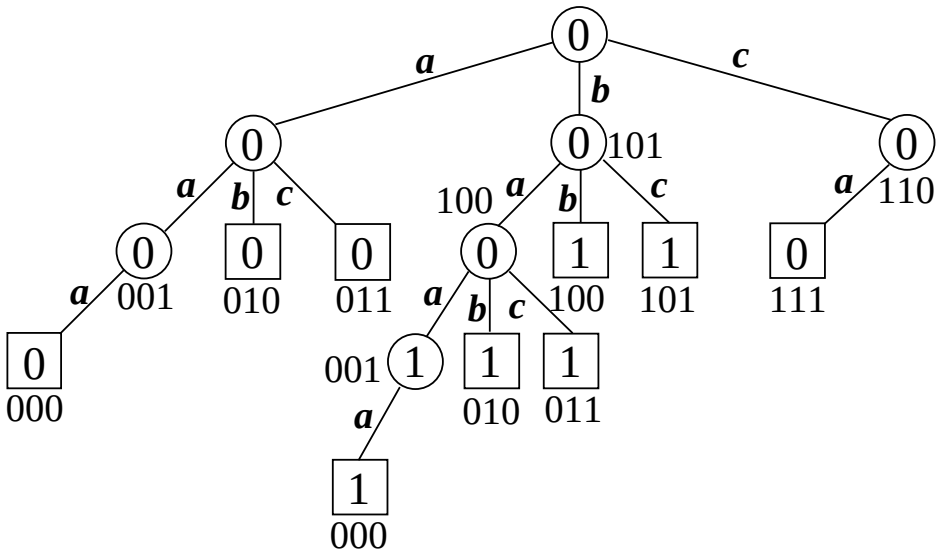

Figure 3.2: An example of VMA tree. This is an integrated tree obtained by multiplexing the trees shown in Figures 2.1 and 2.2. The number written in each node indicates the value of Tn(n).

Algorithm 3.1 Encoding algorithm with a VMA tree T.

Input: A text and parse tree T.

Output: An encoded text.

$n \leftarrow$ the root of T.
$i \leftarrow 0$.
while not end of the input text **do**
    $c \leftarrow$ the next symbol of the input text.
    if there exists child n' of node n with label c and $\mathrm{Tn}(n') \leq i$ **then**
        $n \leftarrow n'$.
    else
        Output $w_i(n)$.
        $i \leftarrow \mathrm{d}(n)$.
        $n \leftarrow$ the root of T.
        $n \leftarrow$ the child of n labeled by c.
    end if
end while

Algorithm 3.2 Constructing a VMA tree.

Input: Alphabet Σ and occurrence probabilities of all characters $\Pr(a_1), \ldots, \Pr(a_{|\Sigma|})$.

Output: Parse tree T.

1: $T \leftarrow$ The tree with root and $|\Sigma|$ children of it.
2: Label the jth edge of T by a_j.
3: **for** $k \leftarrow 0$ to $|\Sigma| - 2$ **do**
4: $m \leftarrow \#\mathcal{N}(S_k)$; $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(\mathcal{T})} \Pr(n)$.
5: **while** $|\Sigma| - d(\hat{n}) - 1 \leq m$ **do**
6: $S_1 \leftarrow$ The average phrase length assuming that we call **Procedure** COMPLETE in Algorithm 3.3.
7: $S_2 \leftarrow$ The average phrase length assuming that we call **Procedure** FINDOPTPOS in Algorithm 3.4 $|\Sigma| - d(\hat{n}) - 1$ times.
8: **if** $S_1 \geq S_2$ **then**
9: Call **Procedure** COMPLETE in Algorithm 3.3.
10: **else**
11: Call **Procedure** FINDOPTPOS in Algorithm 3.4 $|\Sigma| - d(\hat{n}) - 1$ times.
12: **end if**
13: $\hat{n} \leftarrow \mathrm{argmax}_{n \in \mathcal{N}(T)} \Pr(n)$; $m \leftarrow m - |\Sigma| + d(\hat{n}) + 1$.
14: **end while**
15: Call FindOptPos m times.
16: $i \leftarrow 0$.
17: **for all** $n \in \mathcal{N}(D)$ **do**
18: $w_k(n) \leftarrow i$; $i \leftarrow i + 1$.
19: **end for**
20: $S \leftarrow$ The subtree corresponding the node traversed from the root by $a_{|\Sigma|+1}$.
21: $R \leftarrow T$ except for S; Add S to T_{V}; $T \leftarrow R$.
22: **end for**
23: Label the jth edge of each node by a_j $(j = 1, \ldots, |\Sigma|)$.
24: **return** T.

Algorithm 3.3 Procedure COMPLETE.

Input: Parse tree T.

1: $\hat{n} \leftarrow \text{argmax}_{n \in \mathcal{N}(T)} \Pr(n)$.

2: **for** $j \leftarrow 0$ to $|\Sigma|$ **do**

3: Add $|\Sigma| - d(\hat{n})$ children to $\hat{n}$.

4: **end for**

5: $\text{Tn}(n_j) \leftarrow \text{cod}(\text{first}(n_j)) - 1$.

Algorithm 3.4 Procedure FINDOPTPOS.

Input: Parse tree T.

1: $\tilde{n} \leftarrow \text{argmax}_{n \in \mathcal{N}(T)} \Pr(n) \Pr(a_{d(n)+1})$.

2: Add the $(d(\tilde{n}) + 1)$th child n of $\tilde{n}$.

3: $\text{Tn}(n) \leftarrow \text{cod}(\text{first}(n)) - 1$.

3.2 Bound Analysis of VMA Tree

In this section, we discuss the upper and lower bounds of the number of nodes in VMA tree. Let M be the number of codewords, i.e., $M = 2^\ell$ for the codeword of length ℓ. We prove that the upper and lower bounds of the total number of nodes in VMA tree are respectively $M|\Sigma| - |\Sigma|^2/2 + |\Sigma|/(|\Sigma| - 1)$ in Theorem 3.2 and $M \ln(|\Sigma| + 1) - M/2 + 1$ in Theorem 3.3 and that the lower and upper bounds of the number of nodes reduced by integrating multiple parse trees into one are respectively $|\Sigma|^2/2 + 5|\Sigma|/2 - 7$ in Theorem 3.4 and $[|\Sigma|M(|\Sigma| - 1/2) + (|\Sigma| - 6)(|\Sigma| + 1)/2 - |\Sigma|M \ln(|\Sigma| + 1)]/(|\Sigma| - 1)$ in Theorem 3.5.

Theorem 3.2. *An upper bound of the total number of nodes in the VMA tree is* $M|\Sigma| - |\Sigma|^2/2 + |\Sigma|/(|\Sigma| - 1)$ *where* M *and* $|\Sigma|$ *respectively denote the number of codewords and the alphabet size.*

Proof. The leftmost subtrees $S_{i+1}^{(i)}$ for $i = 0 \ldots |\Sigma| - 2$ and tree $T_{|\Sigma|-2}$ of the AIVF tree are left in the VMA tree. Subtrees $S_{i+1}^{(i)}$ have $M - (|\Sigma| - i - 1)$ codewords because each subtree has at least one codeword. A $|\Sigma|$-ary tree with M leaves has at most $(M-1)/(|\Sigma| - 1)$ complete internal nodes. Therefore, the upper bound of the number of nodes in the VMA tree is

$$\begin{aligned}
& \sum_{i=0}^{|\Sigma|-3} \left(\#\mathcal{C}\left(S_{i+1}^{(i)}\right) + \#\mathcal{N}\left(S_{i+1}^{(i)}\right)\right) + \#\mathcal{C}\left(S_{|\Sigma|-1}^{(|\Sigma|-2)}\right) + \#\mathcal{C}\left(S_{|\Sigma|}^{(|\Sigma|-2)}\right) \\
& \qquad + \#\mathcal{N}\left(S_{|\Sigma|-1}^{(|\Sigma|-2)}\right) + \#\mathcal{N}\left(S_{|\Sigma|}^{(|\Sigma|-2)}\right) + 1 \\
\leq & \sum_{i=0}^{|\Sigma|-3} \left(M - (|\Sigma| - i - 1) + \frac{M - (|\Sigma| - i - 1) - 1}{|\Sigma| - 1} \right) + \frac{M-1}{|\Sigma| - 1} + 1 + M \\
= & \; M|\Sigma| - \frac{1}{2}|\Sigma|^2 + \frac{|\Sigma|}{|\Sigma| - 1}.
\end{aligned}$$

□

Theorem 3.3. *A lower bound of the total number of nodes in the VMA tree is* $M \ln(|\Sigma| +$

$1) - M/2 + 1$ *where* M *and* $|\Sigma|$ *respectively denote the number of codewords and the alphabet size.*

Proof. Since the symbols $a_1, \ldots, a_{|\Sigma|}$ are sorted in descending order of their occurrence probabilities, we have $\#\mathcal{N}(S_j^{(i)}) \geq \#\mathcal{N}(S_{j+1}^{(i)})$. Therefore, we have $\#\mathcal{N}(S_{i+1}^{(i)}) \geq M/(|\Sigma| - i)$. Hence the lower bound of the number of nodes in the VMA tree is

$$\begin{aligned}
& \sum_{i=0}^{|\Sigma|-3} \#\mathcal{N}\left(S_{i+1}^{(i)}\right) + \#\mathcal{N}\left(S_{|\Sigma|-1}^{(|\Sigma|-2)}\right) + \#\mathcal{N}\left(S_{|\Sigma|}^{(|\Sigma|-2)}\right) + 1 \\
\geq & \sum_{i=0}^{|\Sigma|-3} \left(\frac{M}{|\Sigma| - i}\right) + M + 1 \\
\geq & M \int_1^{|\Sigma|+1} \frac{1}{i} di - \frac{M}{2} + 1 \\
\geq & M \ln(|\Sigma| + 1) - \frac{M}{2} + 1.
\end{aligned}$$

□

Theorem 3.4. *An upper bound of the number of nodes reduced by the integration is* $[|\Sigma|M(|\Sigma| - 1/2) + (|\Sigma| - 6)(|\Sigma| + 1)/2 - |\Sigma|M \ln(|\Sigma| + 1)]/(|\Sigma| - 1)$ *where* M *and* $|\Sigma|$ *respectively denote the number of codewords and the alphabet size.*

Proof. Subtrees $S_2^{(0)}, \ldots, S_{|\Sigma|}^{(0)}, S_3^{(1)}, \ldots, S_{|\Sigma|}^{(1)}, \ldots, S_{|\Sigma|-1}^{(|\Sigma|-3)}, S_{|\Sigma|}^{(|\Sigma|-3)}$ and $|\Sigma| - 2$ root nodes are reduced by the integration. Analogous to the proof of Theorem 3.3, the upper bound is

$$\begin{aligned}
& \sum_{i=0}^{|\Sigma|-3} \left[\sum_{j=i+2}^{|\Sigma|} \left(\#\mathcal{N}\left(S_j^{(i)}\right) + \#\mathcal{C}\left(S_j^{(i)}\right) \right) \right] + |\Sigma| - 2 \\
= & \sum_{i=0}^{|\Sigma|-3} \left((|\Sigma| - 1)\frac{M}{|\Sigma| - i} + \frac{\frac{M}{|\Sigma|-i}(|\Sigma| - i - 1) - 1}{|\Sigma| - 1} \right) + |\Sigma| - 2 \\
\leq & \frac{|\Sigma|M(|\Sigma| - \frac{1}{2}) + \frac{1}{2}(|\Sigma| - 6)(|\Sigma| + 1) - |\Sigma|M \ln(|\Sigma| + 1)}{|\Sigma| - 1}.
\end{aligned}$$

□

Theorem 3.5. *A lower bound of the number of nodes reduced by the integration is* $|\Sigma|^2/2 + 5|\Sigma|/2 - 7$ *where* $|\Sigma|$ *denotes the alphabet size.*

Proof. Analogous to the proofs of Theorems 3.2 and 3.4, the lower bound is

$$\begin{aligned} & \sum_{i=0}^{|\Sigma|-3} \left[\sum_{j=i+2}^{|\Sigma|} \#\mathcal{N}\left(S_j^{(i)}\right) + 1 \right] \\ \geq & \sum_{i=0}^{|\Sigma|-3} (|\Sigma| - i + 2) \\ = & \frac{1}{2}|\Sigma|^2 + \frac{5}{2}|\Sigma| - 7. \end{aligned}$$

□

3.3 Experiments

We implemented Tunstall coding, AIVF coding, and our proposed method. We abbreviate these programs as Tunstall, AIVF, and VMA, respectively. All the programs we used are written in C++ and compiled by g++ of GNU, version 4.6. We embedded the information of structures of parse trees by balanced parentheses [26] into compressed texts. We ran our experiments on a workstation equipped with an Intel Xeon (R) 3.00 GHz CPU with 12 GB RAM, which operated Ubuntu 12.04. We used "dna" and english.300MB, which is the first 300 MB of "english" from "Pizza&Chili Corpus[2]." For details, please refer to Table 3.1.

First, we measured the compression ratios, compression times, and decompression times of Tunstall, AIVF, and VMA. We measured (compressed file size)/(original file size) as the compression ratio. Table 3.2 shows the compression ratios. The compression ratios of AIVF and VMA are better than that of Tunstall on english.300MB and dna. Since AIVF and VMA essentially perform the same compression, their compression ratios are almost the same. However, slight differences are caused by the differences

[2]http://pizzachili.dcc.uchile.cl/index.html

in the size of parse trees. The compression times are shown in Table 3.3. The compression of VMA and AIVF is slower than that of Tunstall, because VMA and AIVF create more nodes than does Tunstall. In addition, the compression of VMA is faster than that of AIVF where codeword length is long, because VMA creates fewer nodes than does AIVF. The improvement in compression time is larger on english.300MB than on dna. AIVF constructs $|\Sigma| - 1$ parse trees where $|\Sigma|$ is the alphabet size. Therefore, VMA reduces many nodes when a corpus whose alphabet size is large is compressed. Although the number of nodes in the parse tree exponentially grows as the codeword length increases, the compression time of Tunstall does not seem to slow down. The reason is considered to be that the total amount of I/O time is reduced because the total amount of compressed data is decreased as the codeword length increases. Finally, the decompression times are shown in Table 3.4. The decompression of VMA and AIVF is slower than that of Tunstall. Since the number of nodes in the parse trees of VMA and AIVF is larger than that of Tunstall, the reconstruction of the parse trees of VMA and AIVF is slower than that of Tunstall. Moreover, the decompression of VMA is slightly slower than that of AIVF, because VMA needs extra operations to compute the degree of a node by comparing the minimum tree number of its children and the current tree number while decoding.

We also compared seven compression algorithms: Tunstall, AIVF, VMA, *v2vdc* [4], gzip, bzip2, and LZMA. We used the default options for gzip, bzip2, and LZMA. We selected 14 for the codeword length of VF coding methods. The results are shown in Table 3.5. It should be noted that *v2vdc* is a word-based compression method and it therefore is not effective for DNA data. We represent this by "N/A" in Table 3.5. The compression ratios of VMA, AIVF, and Tunstall are worse than those of *v2vdc*, gzip, and bzip2 for the English text, because they do not assume a Markov source. On the other hand, VMA and AIVF are rather good for the DNA text. *V2vdc* takes a long time to compress English text to construct suffix arrays. Although the compression and

decompression speeds of VMA is slower than bzip2 for the English text, those of VMA for DNA text is faster than those of bzip2. The reason is that our implementation for VMA, AIVF, and Tunstall takes $O(|\Sigma|)$ time to traverse the parse tree for a character.

3.4 Chapter Summary

In this chapter, we presented an efficient algorithm for AIVF coding, which integrates the multiple parse trees of an AIVF coding into one, called a VMA tree, and simulates the encoding process and the decoding process on it. We also estimated the number of nodes in the VMA tree. We conducted several experiments to evaluate the compression performance of three compression methods: Tunstall coding, AIVF coding, and VMA coding. The compression ratios of VMA/AIVF coding methods are better than those of Tunstall coding by 20% on natural language texts. Although the decompression of VMA coding is slower than that of AIVF, the compression of VMA coding is up to six times faster than that of AIVF coding.

Although the methods presented in [13, 20] have better compression ratios than AIVF coding, they need to construct suffix trees [9] in order to construct a parse tree, and thus, they take a long time to achieve this. Therefore, from the viewpoint of speeding up compressed pattern matching, AIVF coding with the VMA tree is a strong candidate as well, because usually a parse tree has to be constructed in order to conduct compressed pattern matching on compressed texts of VF coding.

Table 3.1: Experimental text files.

Texts	size (byte)	$\|\Sigma\|$	Content
english.300MB	300000000	225	English document
dna	403927746	16	DNA sequence

Table 3.2: Compression ratios of Tunstall, AIVF, and VMA in percentage.

codeword length	english.300MB			dna		
	Tunstall	AIVF	VMA	Tunstall	AIVF	VMA
8	100.00	79.52	79.52	35.22	26.29	26.29
9	94.19	63.09	63.08	34.23	26.11	26.11
10	93.18	59.98	59.96	32.74	25.91	25.91
11	83.81	59.06	59.03	32.10	25.75	25.75
12	81.24	58.99	58.95	31.09	25.70	25.71
13	78.55	58.64	58.55	30.54	25.64	25.65
14	76.76	58.56	58.38	30.09	25.57	25.59
15	74.37	58.72	58.36	29.65	25.48	25.53
16	73.82	59.20	58.49	29.26	25.52	25.61

Table 3.3: Compression times of Tunstall, AIVF, and VMA in seconds.

codeword length	english.300MB			dna		
	Tunstall	AIVF	VMA	Tunstall	AIVF	VMA
8	19.54	20.26	28.90	18.54	18.45	20.31
9	20.49	20.23	28.26	17.21	18.36	20.42
10	19.65	22.35	29.34	16.62	18.84	21.54
11	19.63	26.31	32.33	17.19	19.22	21.01
12	17.29	34.44	33.17	17.34	20.06	21.37
13	18.50	63.16	39.61	17.21	22.31	23.27
14	19.03	160.71	60.06	16.83	32.48	30.41
15	18.19	642.42	143.83	16.52	96.45	81.04
16	17.69	5241.48	838.33	16.05	774.81	483.77

Table 3.4: Decompression times of Tunstall, AIVF, and VMA in seconds.

codeword length	english.300MB			dna		
	Tunstall	AIVF	VMA	Tunstall	AIVF	VMA
8	17.15	15.61	18.42	8.44	14.40	15.15
9	15.40	14.54	17.29	7.77	14.35	15.11
10	14.36	14.65	17.41	6.90	14.14	14.88
11	12.71	15.14	17.62	6.89	14.59	15.05
12	11.37	15.82	17.94	6.20	14.30	14.93
13	10.90	18.52	19.08	6.04	14.37	15.72
14	9.70	19.53	20.72	5.51	15.16	15.96
15	9.14	23.40	27.77	5.57	16.89	20.36
16	11.78	27.49	35.15	5.86	20.21	25.11

Table 3.5: Experimental results compared with variable length encoding methods. The codeword length of VF codes is fixed to 14.

	comp. ratios (%)		comp. times (sec)		decomp. times (sec)	
	english.300MB	dna	english.300MB	dna	english.300MB	dna
VMA	58.38	25.59	60.06	30.41	20.72	15.96
AIVF	58.56	25.57	160.71	32.48	19.53	15.16
Tunstall	76.76	30.09	19.03	16.83	9.70	5.51
v2vdc	52.61	N/A	6163.93	N/A	4.93	N/A
gzip	37.82	28.12	23.60	49.87	2.79	3.33
bzip2	28.03	25.76	38.10	52.51	13.00	20.98
LZMA	24.98	22.73	345.05	730.48	5.72	7.66

Chapter 4

Dictionary Training Algorithm for Efficient VF Coding

In this chapter, we present a way of training the parse tree by compressing the input text and modifying the parse tree repeatedly. We also discuss a method that uses parts of the input text for training in order to reduce the training time. The training method improves the compression ratio of VF coding rapidly to the level of state-of-the-art compression methods. This work has already been partially presented in [45].

4.1 Reconstruction Algorithm

In this section, we present an algorithm of reconstructing a parse tree to improve the compression ratio. The basic idea is to exchange useless strings in the current parse tree for the other strings not in the parse tree which are expected to be frequently used. Although we must evaluate each string by some measures for doing that, it is quite hard to evaluate precisely in advance as we stated in Chapter 1. Therefore, we employ a greedy approach; we reconstruct the parse tree with two empirical measures: the *accept count* and the *failure count*. For any string s in the parse tree, the accept

count of s, denoted by $A(s)$, is defined as the number of the occurrences of string s 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 times that the prefix $t[1..|t|-1]$ of t was in the parse tree and the codeword traversal failed at the last character of t. If $F(t)$ is sufficiently large, it is expected that we can make the average phrase length longer by including t in the parse tree. The computations of $A(s)$ and $F(t)$ are embedded 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 simultaneously. Figure 4.1 shows an example of computing these measures.

The reconstruction algorithm is shown in Algorithm 4.1. Comparing the minimum of $A(s)$ and the maximum of $F(t)$, the reconstruction algorithm repeats exchanging s for t if the former is less than the latter, that is, it removes s from the parse tree and enter t instead. Note that a reconstructed parse tree is not a complete tree any longer, even if the origin is a complete tree like the Tunstall tree. Several internal nodes might be assigned codewords; thus a coding with such a tree becomes almost instantaneous encoding. To train a parse tree we apply the algorithm many times. For each iteration, it first encodes the input data 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.

Next we discuss the time and space complexities of the algorithm in Algorithm 4.1. We assume that the sets D and E are realized by priority queues to calculate Lines 14 and 15 efficiently. For the loop in Lines 2–11, Line 3 takes $O(|T|)$ time totally, and all lines except Lines 3 and 7 are done in $O(1)$ time for each. Let E' be the number of parsed phrases, namely, it is equal to $|E|$ after processing the loop. Then, the number of iterations of the loop is $O(E')$, and Line 7 takes $O(E' \log E')$ time totally. For the while loop in Lines 13–23, Lines 18 and 22 take $O(\log |D|)$ and $O(\log E')$, respectively. For each iteration, $|E|$ decreases exactly 1 while $|D|$ decreases 1 at most. If $|E| < |D|$

Algorithm 4.1 Reconstruction algorithm for parse trees.

Input: Text T and set of strings in the parse tree D.

Output: A set of strings.

1: $i \leftarrow 1, E \leftarrow \emptyset$.
2: **while** $i < n$ **do**
3: $p \leftarrow$ the longest prefix $T[i..j]$ of $T[i..n]$ which is also included in D.
4: $A(p) \leftarrow A(p) + 1$.
5: **if** $j < |T|$ **then**
6: $q \leftarrow p \cdot T[j+1]$.
7: $E \leftarrow E \cup \{q\}$.
8: $F(q) \leftarrow F(q) + 1$.
9: **end if**
10: $i \leftarrow j + 1$.
11: **end while**
12: $N \leftarrow \emptyset$.
13: **while** $D \neq \emptyset$ and $E \neq \emptyset$ **do**
14: $s \leftarrow \mathrm{argmin}_{s \in D} A(s)$.
15: $t \leftarrow \mathrm{argmax}_{t \in E} F(t)$.
16: **if** $A(s) < F(t)$ **then**
17: $N \leftarrow N \cup \{t\}$.
18: $D \leftarrow D \setminus \{s\}$.
19: **else**
20: **break**
21: **end if**
22: $E \leftarrow E \setminus \{t\}$.
23: **end while**
24: **return** $D \cup N$.

then the number of iterations is just $|E|$, otherwise it is also restricted to $O(|E|)$. Thus, the number of iterations of the while loop is $O(E')$. Therefore, the time complexity of the algorithm is $O(E' \log |D| E' + E' \log E' + |T|)$. Roughly speaking, it is $O(|T| \log |T|)$ since both E' and $|D|$ are $O(|T|)$. For the space consumption, we can prove that it is $O(|T|)$ space from the same discussion on Algorithm 2.8.

4.2 Speeding-up by Sampling

The reconstruction of parse trees discussed above takes much time if the input text is large, since the algorithm scans the whole text many times. If we train with small parts of the text, we can save the training time. Note that we must scan the whole text once to construct the initial parse tree.

We consider training with a string that consists of several *pieces* randomly selected from the text. Using only one part of the input text T, namely a substring of T, does not work well even if we select a substring randomly for each reconstruction, since the parse tree reconstructed by the above algorithm fits too much on the last selection. Using a set of pieces randomly selected from the whole text works well. Let r be the number of pieces, and p be the length of a piece. For given $r \geq 1$ and $p \geq 1$, we generate a sample text S from T at every reconstruction as follows:

$$S := s_1 \cdots s_r \quad (s_k := T[i_k..i_k + p - 1] \text{ for } 1 \leq k \leq r),$$

where i_k is a start position of a piece satisfying $1 \leq i_k \leq |T| - p + 1$. We select the pieces in a uniform random manner for each k. Then, $|S| = rp$ holds. Note that the compression ratios and speeds depend on $|S|$ and r in addition to the number of training iterations.

4.3 Experimental Results

We have implemented the Tunstall coding and the STVF coding with the training approach stated in Section 4.1, and compared them with BPEX [23], ETDC [7], SCDC [5], gzip, and bzip2. Although ETDC/SCDC are variable-to-variable-length codes, their codewords are byte-oriented and designed for compressed pattern matching. Therefore, we added them in our experiments. We chose 16 as the codeword length of both the STVF coding and the Tunstall coding. Our programs are written in C++ and compiled by g++ of GNU, version 3.4. We ran our experiments on an Intel Xeon (R) 3 GHz and 12 GB of RAM, running Red Hat Enterprise Linux ES Release 4.

We used DNA data, XML data, English texts, and Japanese texts to be compressed (see Table 4.1). GBHTG119 is a collection of DNA sequences with meta data in GenBank[1], from which we extracted only DNA part. DBLP2003 consists of all the data in the year 2003 from dblp20040213.xml[2]. Reuters-21578 (distribution 1.0)[3] is a test collection of English texts. Mainichi1991[4] is from Japanese news paper, *Mainichi-Shinbun*, in the year 1991.

4.3.1 Compression Ratios and Speeds

The methods in our experiments are the following nine: Tunstall (the Tunstall coding without training), STVF (the STVF coding without training), Tunstall-100 (the Tunstall coding with 100 times training), STVF-100 (the STVF coding with 100 times training), BPEX, ETDC, SCDC, gzip, and bzip2.

Figure 4.2 shows the results of compression ratios, where every compressed data include the dictionary information. We indicate the compression ratios of the averages

[1] `http://www.ncbi.nlm.nih.gov/genbank/`

[2] `http://www.informatik.uni-trier.de/~ley/db/`

[3] `http://www.daviddlewis.com/resources/testcollections/reuters21578/`

[4] `http://www.nichigai.co.jp/sales/corpus.html`

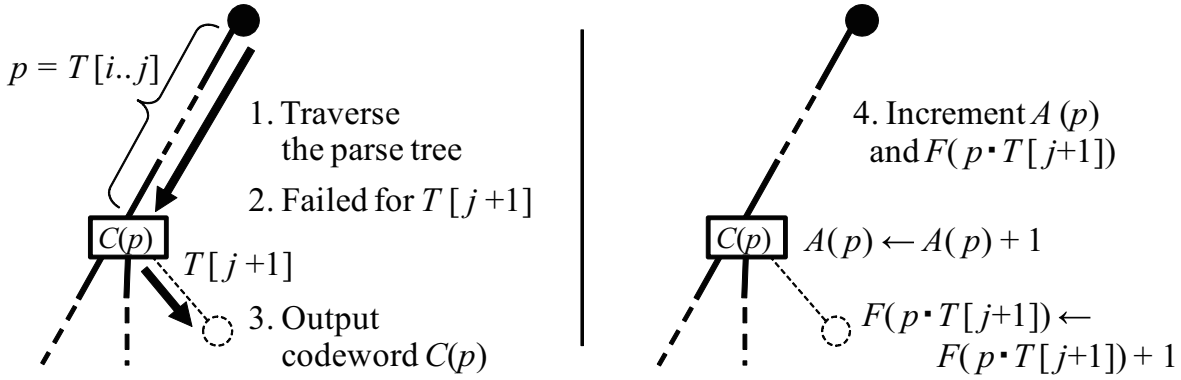

Figure 4.1: An example of computing accept counts and failure counts.

Table 4.1: Outline of the text files used for our experiments.

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)

of ten executions for Tunstall-100 and STVF-100. STVF, Tunstall-100, and STVF-100 were the best in the compression ratio comparisons for GBHTG119. Since ETDC and SCDC are word based compression methods, they did not work well for the data that are hard to divide into words, such as DNA sequences and Japanese texts. Note that, while Tunstall had no advantage to STVF, Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were better than gzip. Figure 4.3 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 4.4 shows the results of decompression times. The decompression times of ETDC, SCDC, and gzip are the shortest, and those of bzip2 and BPEX are the longest. The results of Tunstall and STVF were between those of BPEX and ETDC/SCDC in all the data. The decompression procedures of Tunstall-100 and STVF-100 take more time than that of Tunstall and STVF.

4.3.2 Effects of Training

We examined how many times we should apply the reconstruction algorithm for sufficient training. We chose Reuters-21578 as the test data in the experiments. Figure 4.5 shows the result of the effect of training for STVF and Tunstall. The compression ratios of both algorithms were improved rapidly as the number of reconstruction increases. They seem to come close asymptotically to the same limit, which is about 32%.

We also examined how the sampling technique stated in Section 4.2 effects on compression ratios and speeds. Figures 4.6 and 4.7 show the results for the Tunstall codes with 20 times training. Figure 4.6 shows compression ratios and Figure 4.7 does compression speeds. We measured the average of 100 executions for each result. We observed that the compression ratio achieves almost the same as that of the training method without sampling when the sample size $|S|$ is 25% of the entire text and the

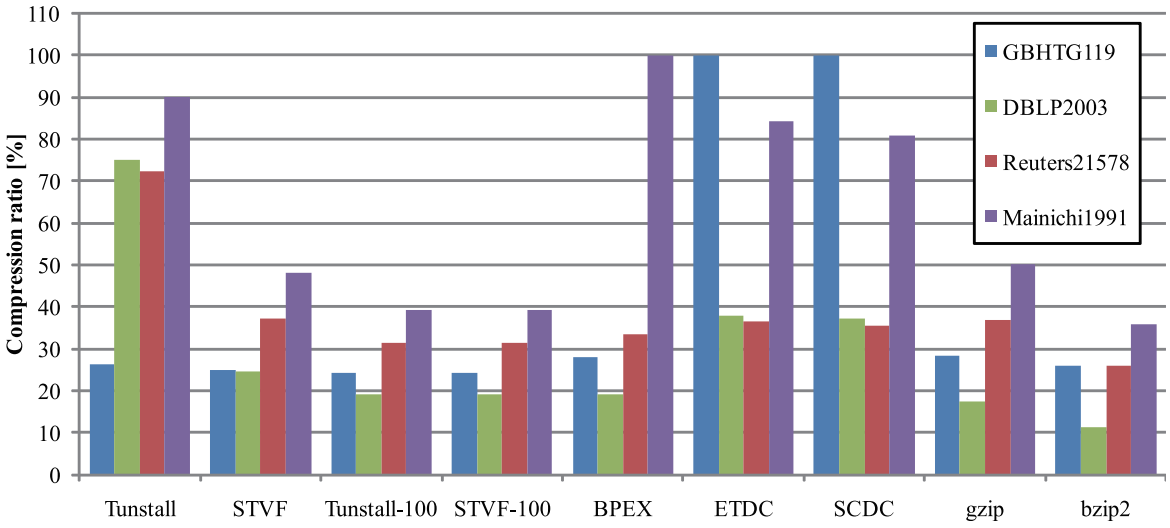

Figure 4.2: Compression ratios.

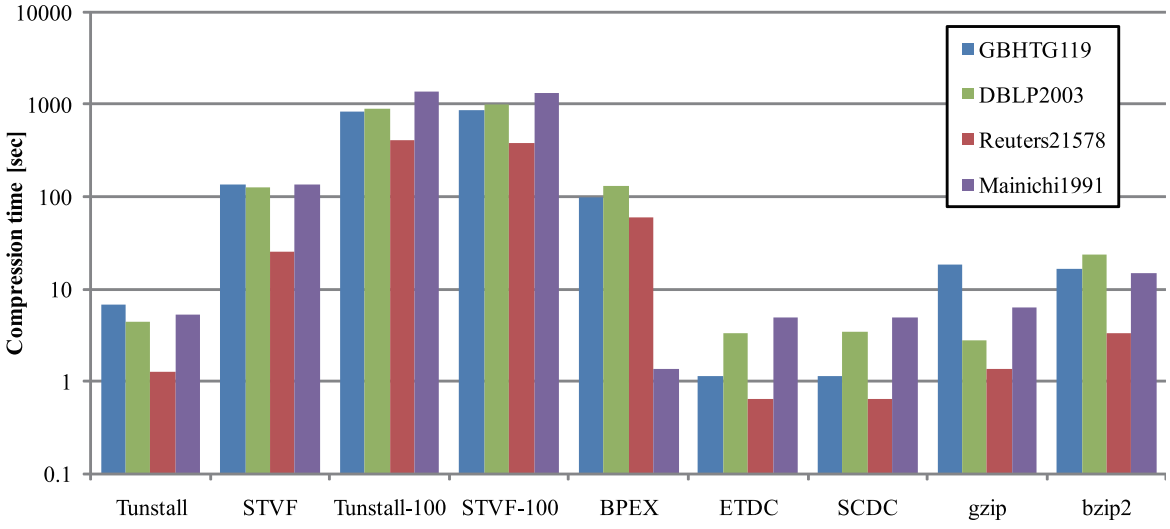

Figure 4.3: Compression times.

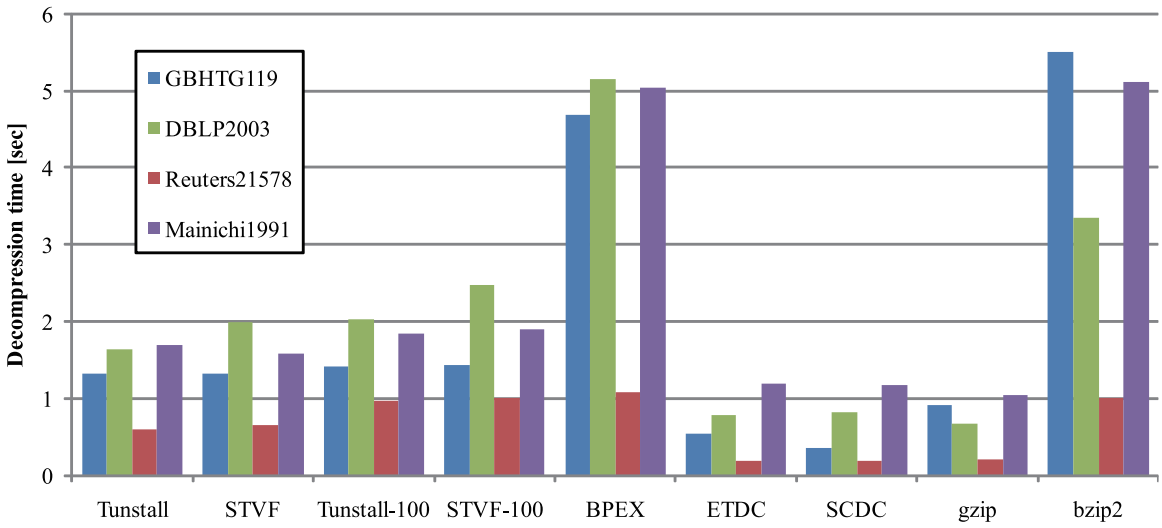

Figure 4.4: Decompression times.

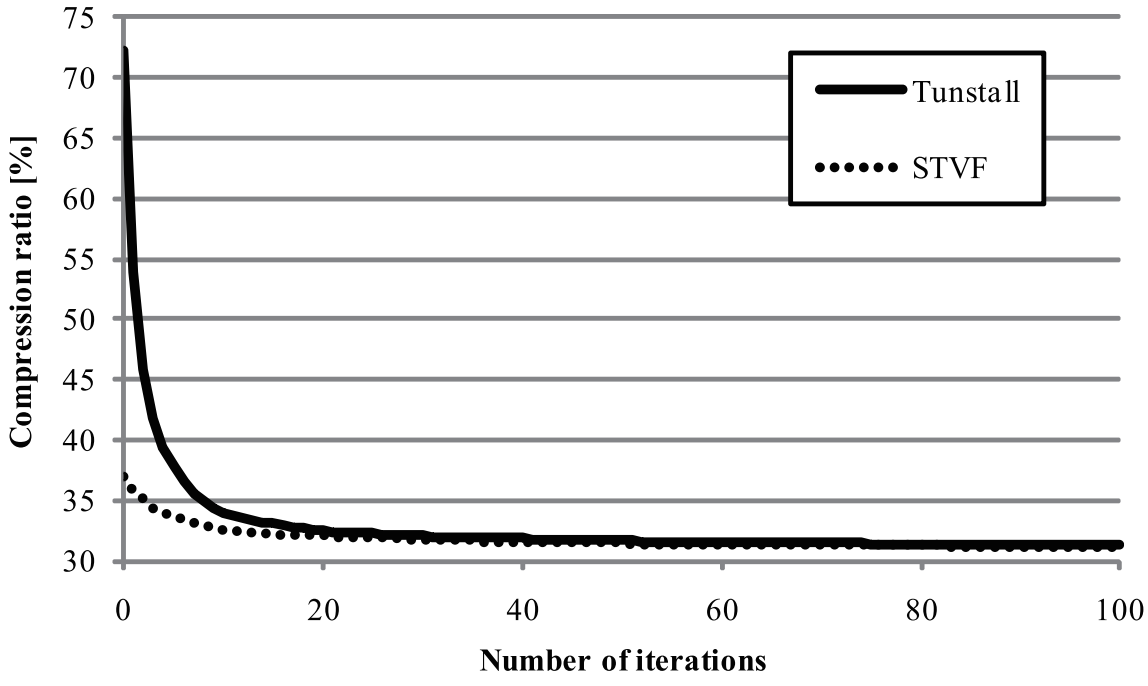

Figure 4.5: The effects of training.

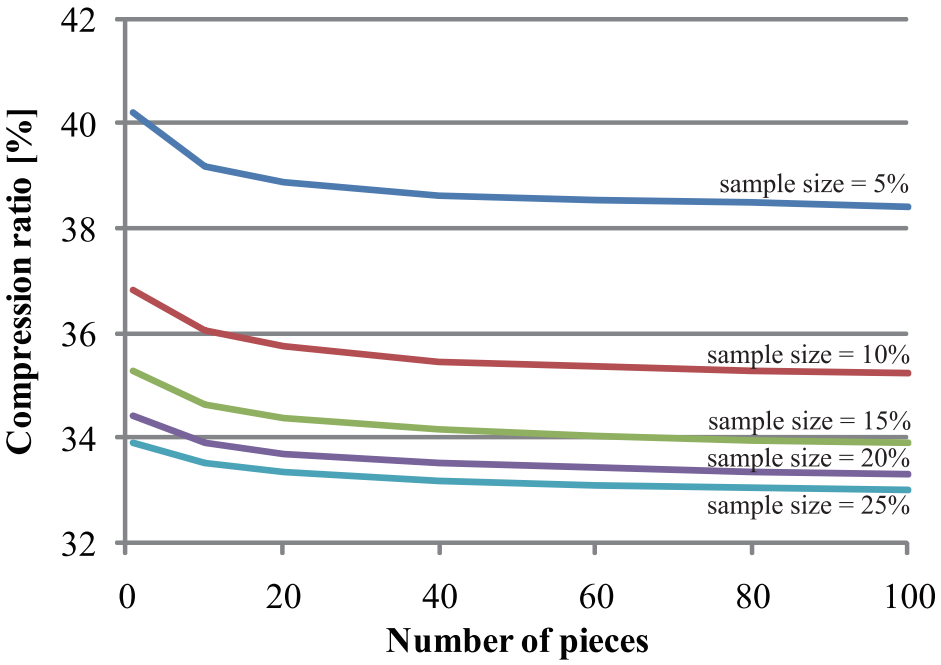

Figure 4.6: Compression ratio of training method with sampling.

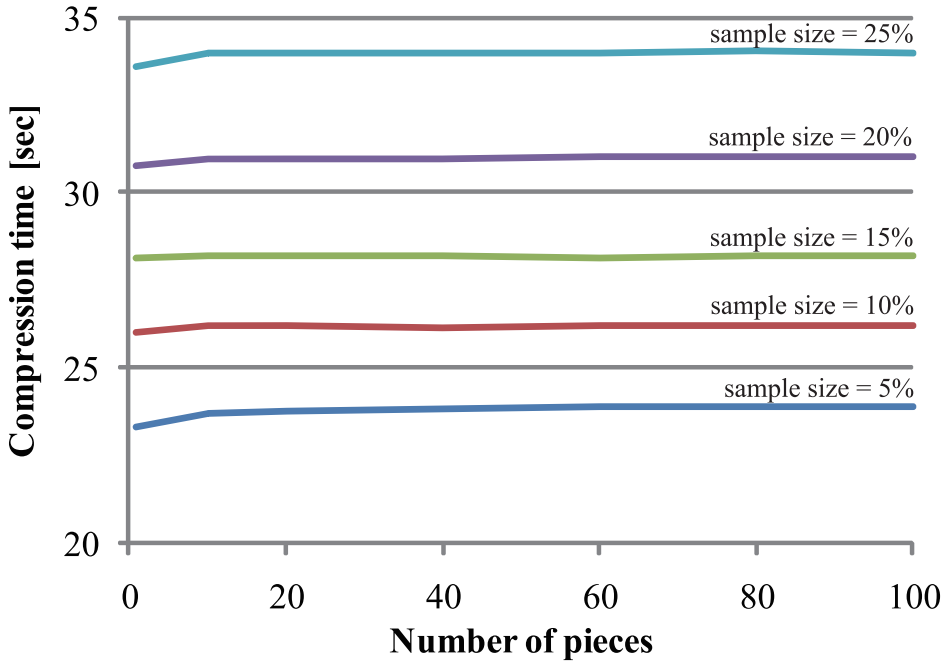

Figure 4.7: Compression time of training method with sampling.

number r of pieces is 100. The Tunstall code with training is superior to BPEX in compression ratios when $|S|$ is 20% and $r = 40$. The average compression time of the Tunstall codes at that point was 30.97 seconds, while BPEX takes 58.77 seconds.

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

4.4 Chapter Summary

In this chapter, we presented a method of improving VF coding by training the parse tree and carried out some experiments for evaluating it. The experimental results showed that our method improves compression ratios of VF coding to the level of state-of-the-art compression methods, such as gzip and BPEX. Tunstall coding with training are about twice faster than that of BPEX in compression speed when we gain almost the same compression ratios. VF coding methods with training are stable and are widely applicable to various data: not only English language texts, but also Japanese texts, DNA data, and so on.

Chapter 5

Efficient VF Coding Algorithm Using Re-Pair Algorithm

In this chapter, 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 [21]. 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 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.

5.1 Re-Pair Algorithm

The Re-Pair algorithm 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 $\Sigma := \{a_1, \ldots, a_{|\Sigma|}\}$, $V := \{a_{|\Sigma|+1}, \ldots, a_{|\Sigma|+|V|}\}$, $\sigma \in V$, and R are the terminal alphabet, the non-terminal alphabet, the start symbol, and a finite relation from V to $(\Sigma \cup V)^*$, respectively. Note that Σ and V are disjoint sets. The CFG constructed by the Re-Pair algorithm consists of rules in which

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

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

Algorithm 5.1 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 (See Figure 5.1). After that, the algorithm adds the start symbol σ, which generates the obtained sequence, into R. Finally, the algorithm encodes all the rules except for σ by *chiastic slide method*, and encodes σ with *minimum-redundancy codes* [25, 39].

It is shown by Larsson and Moffat that Re-Pair runs in $O(n)$ time for an input text of length n. To achieve $O(n)$ time processing, the input text is transformed to a set of doubly-linked lists in which the same bigrams are linked. Moreover, a hash table to access a descriptor of each bigram in $O(1)$ time and a priority queue for the descriptors to manage the frequencies of bigrams are also used.

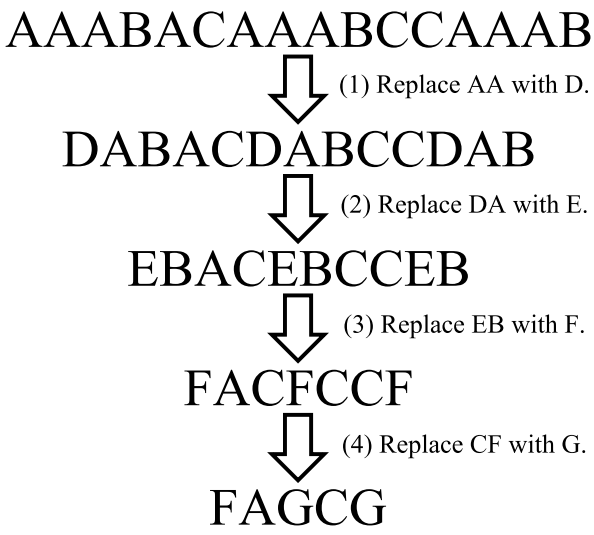

Figure 5.1: Example of Re-Pair algorithm. It repetitively replace the most frequent bigram into a new symbol.

Algorithm 5.1 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: **while** the frequency of the most frequent bigram in T is not equal to 1 **do**
3: $(\beta, \gamma) \leftarrow$ the most frequent bigram in T.
4: Add $(a_s \Rightarrow \beta\gamma)$ to R.
5: Replace all the bigrams $\beta\gamma$ in T with a_s by the left-to-right manner.
6: $s \leftarrow s + 1$.
7: **end while**
8: Add $(\sigma \Rightarrow T)$ to R.
9: **Output** encoded R with an entropy encoding.

5.2 Re-Pair-VF

We can easily encode the rule set generated by the Re-Pair algorithm using a fixed-length code so a_i is coded by a $\lceil \lg \xi \rceil$-bits integer, where ξ denotes the number of non-terminal symbols and 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 ξ, we can obtain the minimum output in this framework. Note that ξ 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 a_i is encoded into a $\lceil \lg \xi \rceil$-bits integer. We output $\xi - |\Sigma|$ bigrams in addition to the information of Σ as the dictionary, where the dictionary size is $2(\xi - |\Sigma|)\lceil \lg \xi \rceil$ bits, as well as some auxiliary bits for storing Σ and ξ. 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 $|\sigma|\lceil \lg \xi \rceil$ bits as the encoded sequence, where $|\sigma|$ is the length of the right-hand side of σ. Therefore, the estimated output size $f(\xi)$ is given as follows:

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

where σ_ξ is the sequence corresponding to the initial symbol with a dictionary size of ξ (see Figure 5.2). The term $|\Sigma|$ is an invariant factor and $|\sigma_\xi|$ depends on the number of repetitions, which correspond to the size of the rule set R. This means that f depends

only on ξ. In other words, the value of ξ controls the final output size.

After computing $f(\xi)$ for each intermediate rule set, we can find the best value of ξ for f after all the repetitions are complete. We denote this value ξ as $\hat{\xi}$. 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.

There are two approaches for outputting σ with $\sigma_1, \cdots, \sigma_{m(\hat{\xi})}$ after obtaining $\hat{\xi}$. The first approach is to rewind the rule set constructed using the Re-Pair algorithm to produce the intermediate set for $\hat{\xi}$ and replace T (see Figure 5.3). The second approach is to preserve ξ 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 5.2 shows the first approach.

The function $R(i)$ in Algorithm 5.2 denotes the bigram of the right-hand side of the ith rule a_i. For example, for $(a_i \Rightarrow \beta\gamma) \in R$, $R(i) := (\beta, \gamma)$. In this algorithm, we identify the rule a_i by its subscript i while $\sigma[i]$ denotes the ith non-terminal symbol on the right-hand side of σ.

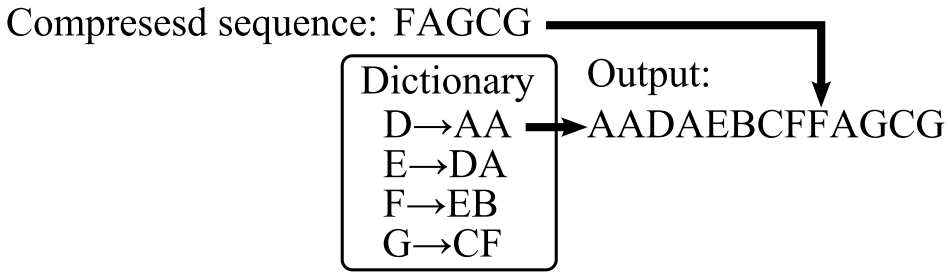

Figure 5.2: Output sequence of our algorithm. We output the right hand side of each entry in the dictionary followed by the sequence corresponding to the initial symbol σ. In this figure, $|\Sigma| = 3$ and $\xi = 7$ hold. The output size is therefore 39 bits.

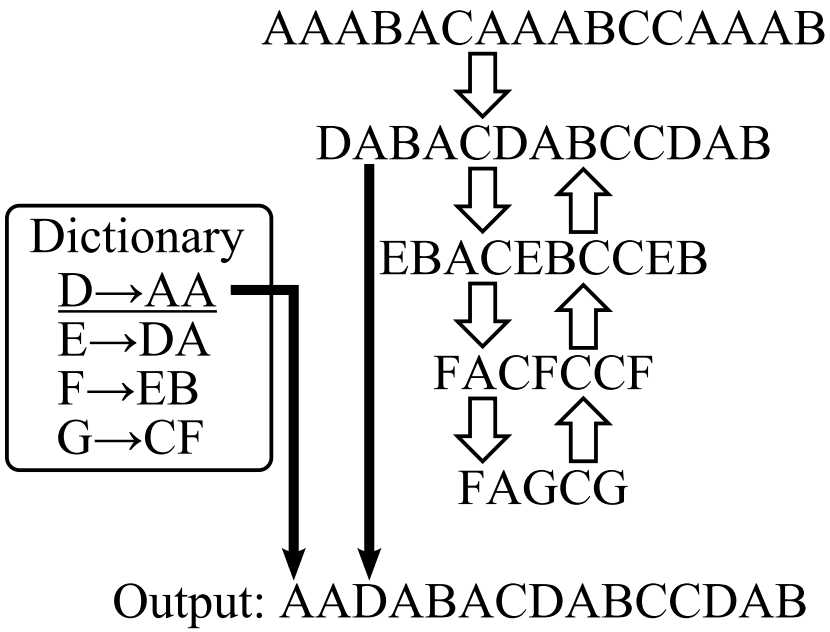

Figure 5.3: The first approach for outputting the compressed sequence and the dictionary after obtaining $\hat{\xi}$. Underlined entries of the dictionary and $\sigma_{\hat{\xi}}$ are output.

Algorithm 5.2 Re-Pair-VF 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: $\xi \leftarrow |\Sigma| + 1$; $R \leftarrow \emptyset$; $b \leftarrow \infty$; $\hat{\xi} \leftarrow \xi$.
2: **while** the frequency of the most frequent bigram in T is not equal to 1 **do**
3: $(\beta, \gamma) \leftarrow$ the most frequent bigram.
4: Add $(a_\xi \Rightarrow \beta\gamma)$ to R.
5: Replace all of the bigrams $\beta\gamma$ in T with a_ξ from left to right.
6: **if** $f(\xi) < b$ **then**
7: $b \leftarrow f(\xi)$.
8: $\hat{\xi} \leftarrow s$.
9: **end if**
10: $\xi \leftarrow \xi + 1$.
11: **end while**
12: Add $(\sigma \Rightarrow T)$ to R.
13: **Output** $\hat{\xi}$ and the information related to Σ.
14: **for** $i \leftarrow |\Sigma| + 1$ **to** $\hat{\xi}$ **do**
15: **Output** $R(i)$ with $\lceil \lg \hat{\xi} \rceil$ bits for each symbol.
16: **end for**
17: **for** $i \leftarrow 1$ **to** the size of the right-hand side of σ **do**
18: **Call Procedure** REWIND-OUTPUT($\sigma[i]$, $\hat{\xi}$, R).
19: **end for**

Algorithm 5.3 Procedure REWIND-OUTPUT.

Input: Integer ξ, the best dictionary size $\hat{\xi}$, and rule set R.

1: **if** $\xi \leq \hat{\xi}$ **then**
2: **Output** ξ with $\lceil \lg \hat{\xi} \rceil$ bits.
3: **else**
4: $(\beta, \gamma) \leftarrow R(\xi)$.
5: **Call Procedure** REWIND-OUTPUT(β, $\hat{\xi}$, R).
6: **Call Procedure** REWIND-OUTPUT(γ, $\hat{\xi}$, R).
7: **end if**

5.3 Experiments

5.3.1 Compression Performance

We implemented our proposed algorithm, known as Re-Pair-VF, and compared it to STVF coding (STVF) [13], the original Re-Pair algorithm[1] (Re-Pair), BPEX [23], gzip, bzip2, and LZMA. We measured the compression ratios and the compression and decompression times. We used the default options 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 5.1 for details). "Dazai.utf.txt" was the complete works of Osamu Dazai[2], which was written in Japanese and encoded by UTF-8. "DBLP2003.xml" comprised all of the 2003 data from dblp20040213.xml[3]. "GBHTG119.dna" was a collection of DNA sequences with meta data in GenBank[4], from which we extracted only DNA part. "Reuters21578.txt" (distribution 1.0)[5] was a sample collection of English texts.

Table 5.2 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.

[1] `http://ihome.cuhk.edu.hk/~b126594/en/restore.html`.

[2] `http://j-texts.com/`.

[3] `http://www.informatik.uni-trier.de/~ley/db/`.

[4] `http://www.ncbi.nlm.nih.gov/genbank/`.

[5] `http://www.daviddlewis.com/resources/testcollections/reuters21578/`.

Table 5.3 shows the maximum size of the dictionary $\max(\xi)$ and the best value $\hat{\xi}$ defined in the previous section. We can observe that $\hat{\xi}$ becomes almost half of $\max(\xi)$ from the result.

Table 5.4 shows the compression times. The results show that Re-Pair-VF was two times faster than Re-Pair. This means that $\hat{\xi}$ can be selected with no increase in the time requirements. Moreover, Re-Pair took longer to encode the rules with complicated methods.

Table 5.5 shows the decompression times. Re-Pair-VF was faster than Re-Pair and STVF, and approximately three times faster than bzip2.

5.3.2 Pattern Matching Performance

We also implemented pattern matching algorithms for Re-Pair-VF according to the methods of Kida *et al.* in 2003 [14] to compare the pattern matching performance with compressed texts using STVF, BPEX, and gzip. 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 5.6–5.9 list 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 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.

Table 5.1: 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 5.2: 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 5.3: The maximum dictionary size (denoted by $\max(\xi)$) and the best size of dictionary (denoted by $\hat{\xi}$).

	$\max(\xi)$	$\hat{\xi}$
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 5.4: Compression times in seconds for each dataset.

	Re-Pair-VF	Re-Pair	STVF	BPEX	gzip	bzip2	LZMA
Dazai.utf.txt	3.536	7.372	1240.854	22.953	0.752	0.820	7.048
DBLP2003.xml	41.339	109.287	1609.241	145.601	2.528	14.925	54.279
GBHTG119.dna	46.959	152.454	1708.855	84.489	17.513	11.561	160.690
Reuters21578.txt	10.881	25.002	1395.139	54.919	1.268	2.416	20.637

Table 5.5: 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

Table 5.6: Matching throughput with DBLP2003.xml (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	202.315	106.427	1506.524	159.630
10	182.975	106.297	1508.410	160.875
15	200.308	106.147	1508.409	163.289
20	199.501	105.899	1508.410	154.789
25	198.872	105.811	1508.409	153.303
30	194.906	105.603	1508.409	151.162
35	196.587	105.318	1508.410	133.802
40	195.231	105.014	1508.410	133.130
45	193.363	104.385	1508.409	131.382
50	191.370	103.897	1508.410	136.392

Table 5.7: 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 5.8: Matching throughput with GBHTG119.dna (MB/s).

m	Re-Pair-VF	STVF	BPEX	gzip
5	159.430	55.352	1083.897	86.110
10	149.633	55.268	1089.604	86.706
15	148.720	55.234	1088.566	86.533
20	153.030	55.181	1089.604	86.762
25	156.903	55.111	1089.604	86.614
30	162.581	54.989	1089.604	86.875
35	165.938	54.947	1089.604	86.646
40	169.492	54.834	1089.604	86.460
45	174.745	54.664	1087.529	86.829
50	177.953	54.524	1089.604	86.795

Table 5.9: 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

5.4 Chapter Summary

In this chapter, 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 coding methods 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.

Chapter 6

Application of VF Coding to Large Texts

In this chapter, we discuss methods for applying VF coding to large texts and their efficiency. We propose a large text compression method for VF coding by block separation and a direct accessing method on a compressed text by VF coding. We additionally give experimental results of each proposed method in this chapter.

6.1 Efficient VF Coding by Block Dividing and Shared Dictionaries

Mainly to memory usage limitation, it is hard for compression systems to handle over hundreds of megabytes of text at once, and thus, an ingenious device is needed. For example, online compression methods such as Lempel-Ziv family compression methods utilize a dictionary window scheme or discard a portion of its dictionary when it increased too much. On the other hand, offline methods such as a compression based on the Burrows-Wheeler transform [8] divide an input text into a sequence of smaller disjoint blocks, which are then compressed.

When the input text is divided into blocks to be compressed by a dictionary-based compression, we expect the compression performance to be improved by sharing a part of dictionary among blocks. Wan and Moffat [42] proposed a series of methods that merge dictionaries extracted by a Re-Pair algorithm proposed by Larsson and Moffat [21]. The Re-Pair algorithm is a simple offline compression method based on grammar transformation, and it has an extremely high compression ratio. In Wan and Moffat's methods, the memory consumption is successfully controlled in practice with keeping high compression ratios. However, the compression speeds are considerably sacrificed.

In this section, we take two simpler approach to this issue. The first approach prepares a common part of dictionary in advance and share it among all blocks. We call this *Re-Use* algorithm. In this approach, the compression speed and ratio depend on three parameters: block size, dictionary size, and size of shared dictionary. We can easily guess that the amount of dictionaries stored for every blocks (*local dictionaries*) is expected to be decreased by increasing the ratio of shared dictionary, whereas the compression ratio for each block will be depressed. The second approach adaptively reconstructs the shared dictionary, called *Adaptive Dictionary Sharing method* (*ADS method*). In ADS method, only useful entries of the dictionary extracted at the preceding block are reused at the next block, i.e., the entries which appear in the next block many times over a given threshold are shared between the consecutive blocks.

Especially, we discuss here the effect of sharing a part of dictionary on compression performances when input text is divided into disjoint blocks to be compressed by the Re-Pair algorithm. In our method, we use a part of the set of rules as a shared dictionary, which is extracted by the Re-Pair algorithm from the first block of text.

To examine the compression performance of our approach, we implemented Re-Use method and ADS method and carried out several experiments using various combinations of parameters. Experimental results showed that our proposed methods ran

much faster than Re-Merge and achieved similar compression ratios to Bzip2 for several combinations of parameters, even though they use fixed-length codewords. They also revealed that Re-Use method was actually effective when the block size is larger, the length of codewords (the dictionary size) is about 20, and approximately half the dictionary is shared for English text and that ADS method yielded better compression ratio than that of Re-Use method.

6.1.1 Re-Use Algorithm

In this section, we make a brief sketch of the block-wise compression scheme we used. Re-Use divides the input text T of length n into blocks of fixed length b and then runs Re-Pair for each block. Moreover, Re-Use shares a part of the dictionary among all blocks, i.e., it seeks to *reuse* useful entries in the dictionary. We call the shared part as *shared dictionary*. Let ℓ and s be the codeword length and the shared dictionary size. Now we assume that b, s, and ℓ are given as input parameters. We also assume that the input text T is represented as a sequence of nonnegative integers $\{0, \ldots, |\Sigma| - 1\}$.

Re-Use has two phases: shared dictionary construction and block-wise compression. In the former, Re-Use constructs a shared dictionary only from a part of the input instead of the whole input. There are several way of choosing the representative part from the input. For example, the most simple way is to choose the first block as the part. Another way is to sample some pieces from the whole input and concatenate them into one for the part. How to choose the representative part affects the compression performance. We will present three ways at the end of this section. For the construction algorithm, the same way of Re-Pair is used, i.e., the most frequent bigrams in the chosen part are replaced with generating rules of a CFG.

After constructing a shared dictionary, Re-Use processes the blocks one by one. The process is performed as follows: (1) for each entry in the shared dictionary, replace the bigrams in the current block with a corresponding non-terminal symbol, and then

obtain an intermediate compressed sequence, (2) apply Re-Pair onto the intermediate sequence to obtain a fully-compressed sequence with a local dictionary, (3) output the fully-compressed sequence and local dictionary, (4) and repeat (1)–(3) for all the blocks. Figure 6.1 illustrates the process flow of Re-Use.

The dictionary and sequence are usually encoded separately. We output the encoded shared dictionary at first, and then output all the encoded local dictionaries. For our experiments in Section 6.3.1, we encode all the symbols in the dictionaries with ℓ bits. Since each entry in the dictionaries consists of two symbols, any delimiters between two adjacent encoded entries are not required. Moreover we have no use to encode the left-hand side for all the entries because they are just ordered in serial number.

Now we discuss how to sample the representative part for constructing the shared dictionary.

First-Block Sampling *First-Block Sampling method* is the most naive way. It applies Re-Pair to the first block $t := T[0 : b)$ of the input to obtain a dictionary. In this method, we reuse the first s entries of the obtained dictionary as the shared dictionary. This method is much lightweight and works well when the type of the input does not change.

Random Sampling *Random Sampling method* randomly takes samples over the whole input, and then concatenates the sampled pieces into one to make the representative sequence (Figure 6.2). The shared dictionary is obtained by running Re-Pair on the sampled text. Let r and p be the number of sampling pieces and the length of each piece, respectively. We assume that these are additionally given as input parameters. As shown in Section 6.3.1, this method works better than First-Block Sampling method when the type of the input changes in the middle. However, it causes the issue of offline because we have to scan the whole input at once in order to gather the pieces.

Block-Head Sampling *Block-Head Sampling method* takes samples only from the head portions with length c of all blocks (Figure 6.3). We assume c is a given parameter. This method is simpler than Random Sampling method, but it works as well. Of course, this also has the offline issue as same as Random sampling for a large input.

6.1.2 Adaptive Dictionary Sharing Method

We now discuss ADS method. In ADS method only the rules which are frequently used in the next block are reused. The rest infrequent rules are overwritten by new rules. Let m be the *minimum frequency*. A rule $\alpha \in V$ is said to be *frequent* if it is used m times or more when replacing the next block. The ith block of the input is denoted by $B[i](0 \leq i < N)$, where $N = \lceil n/b \rceil$ denotes the number of blocks. It should be noted that the dictionary size $|V \cup \Sigma|$ is less than or equal to 2^ℓ.

The outline of our algorithm is as follows (see also Algorithm 6.1).

1. Applying Re-Pair to the first block $B[0]$, we gain an initial dictionary and the compressed data of $B[0]$.

2. We repeat the following process for the remaining blocks $B[i](1 \leq i < N)$.

    2-1. We repeat replacing the current block $B[i]$ by using all the bigrams in dictionary D one by one with checking if it is frequent or not for the given threshold m. If the bigram is not frequent, we leave the replacement undone and mark the bigram with "vacant." We also count the number of marked rules simultaneously.

    2-2. We apply Re-Pair to the intermediate sequence (half-compressed data) obtained from 2-1. In this step, new rules overwrite the "vacant" entries in the dictionary.

    2-3. We output the compressed sequence and the newly created rules with the position information where the overwriting occurs.

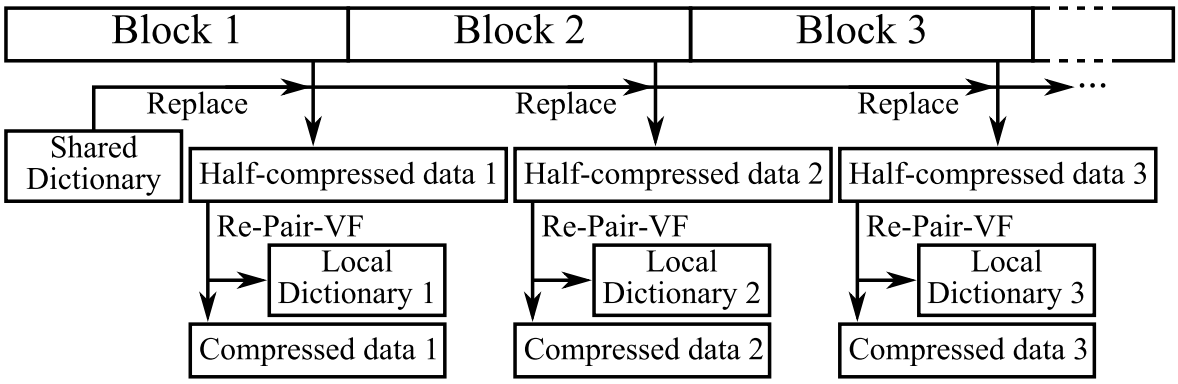

Figure 6.1: The process flow of Re-Use.

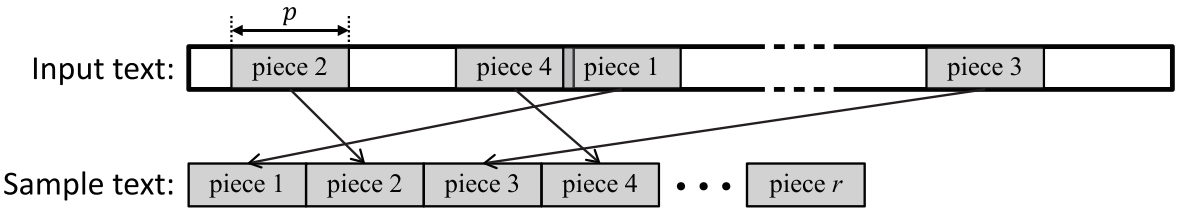

Figure 6.2: Random sampling method.

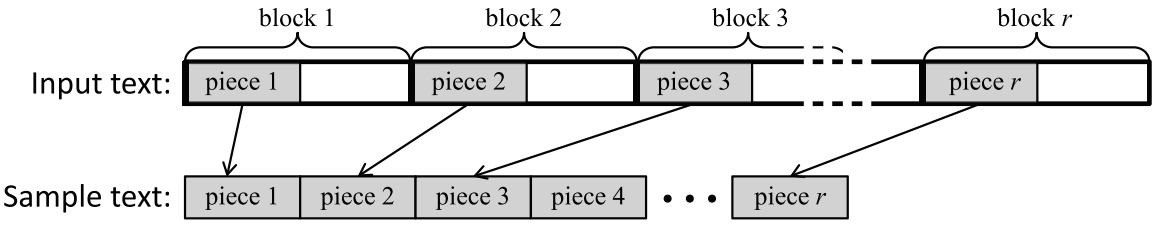

Figure 6.3: Block-Head sampling method.

Note that the rule overwriting in this approach jumbles the dependency ordering of the rules of Re-Pair. Recall that, for Re-Pair, we can assume that any rule $\alpha_i \Rightarrow \alpha_j\alpha_k$ satisfies $j, k < i$. However, the rule overwriting disrupts this ordering. Thus, an additional effort is required to perform correct replacing at the process 2-1.

The replacement is correctly done by processing the rules in the chronological order of generation. We manage the order with three queues $\mathcal{N}$, $\mathcal{F}$, and $\mathcal{U}$ to trace all rules in the order. Queues $\mathcal{N}$, $\mathcal{F}$, and $\mathcal{U}$ respectively store indexes of vacant entries, frequent entries, and used entries. At first, we set $\mathcal{N} := \langle 0, \ldots, 2^\ell - |\Sigma| - 1\rangle$, $\mathcal{F} := \emptyset$, and $\mathcal{U} := \emptyset$ because all the entries are vacant at first. Lines 4–24 process each block. Step 2-1 is performed in Lines 5–13. We dequeue an entry index from queue $\mathcal{U}$ and examine whether the entry is frequent or not one by one. If the entry is frequent, we replace all occurrences of the bigram corresponding to the entry with the entry index and enqueue the entry index into $\mathcal{F}$, otherwise we enqueue the index into $\mathcal{N}$. Step 2-2 is done in Lines 14–20. We dequeue an entry index from $\mathcal{N}$ to replace the most frequent bigram in the current block with the index. After that, the index is enqueued into $\mathcal{F}$. We obtain a fact that all the elements of queue $\mathcal{U}$ are enqueued in the chronological order in Line 9 from following three observations: (1) queue $\mathcal{U}$ is initialized by empty set; (2) entry indexes are enqueued in the order of their generation in Line 19; and (3) queue $\mathcal{U}$ is substituted by $\mathcal{F}$ in Line 21. Therefore, all the elements in $\mathcal{U}$ are arranged in the chronological order in Line 5.

Checking whether each rule is frequent or not can be done in $O(1)$ time by utilizing the data structure proposed in [21]. Thus, our algorithm runs in $O(n)$ time as well as Re-Pair. The same information is not required when decoding, because the sequence of terminal symbols can be restored only by the dictionary.

Algorithm 6.1 ADS algorithm.

Input: Divided text $B[0 : N-1]$ into blocks.

Output: An encoded text.

1: $\mathcal{N} \leftarrow \langle 0, \ldots, 2^{\ell} - |\Sigma| - 1 \rangle$.
2: $\mathcal{F} \leftarrow \emptyset$.
3: $\mathcal{U} \leftarrow \emptyset$.
4: **for all** blocks $B[j]$ **do**
5: **while** $\mathcal{U} \neq \emptyset$ **do**
6: $i \leftarrow$ DEQUEUE($\mathcal{U}$).
7: **if** bigram $D[i]$ occurs m times or more in $B[j]$ **then**
8: Replace all the occurrences of $D[i]$ in $B[j]$ with i.
9: ENQUEUE($\mathcal{F}, i$).
10: **else**
11: ENQUEUE($\mathcal{N}, i$).
12: **end if**
13: **end while**
14: **while** all bigrams in $B[j]$ are not unique **do**
15: $i \leftarrow$ DEQUEUE($\mathcal{N}$).
16: $\alpha \leftarrow$ the most frequent bigram in $B[j]$.
17: $D[i] \leftarrow \alpha$.
18: Replace all the occurrences of α in $B[j]$ with i.
19: ENQUEUE($\mathcal{F}, i$).
20: **end while**
21: $\mathcal{U} \leftarrow \mathcal{F}$.
22: $\mathcal{F} \leftarrow \emptyset$.
23: **Output** $B[j]$.
24: **end for**

6.2 Direct Access on Compressed Texts by VF Coding

In this section, we focus on a problem of extracting a substring from an encoded text, i.e., the problem is to extract $T[pos], \ldots, T[pos+\ell]$ from the encoded text $\mathcal{E}(\mathcal{T})$ for given position *pos* and length ℓ, where T is an uncompressed text. We call the problem as the *substring decompression problem*. A naive approach to the problem decompresses and scans the data from the beginning, thus requires $\Theta(pos + \ell)$ time. Bille *et al.* [3] proposed a grammar-based compression method that solves the problem in $O(\log N + \ell)$ time, where N denotes the length of T. Maruyama *et al.* [22] proposed an excellent online grammar-based compression algorithm, named FOLCA, which solves the problem in $O((\log N + \ell) \log n / \log \log n)$ time, where n denotes the number of variables in the grammar. These methods are sophisticated, but required to load the whole data on memory.

We take a simple approach to the problem, which is to add an auxiliary index structure to VF codes. The index structure stores the phrase boundaries on text T, and we represent the index as a *fully indexable dictionary* (FID). This enables us to access in constant time directly to the codeword that contains $T[pos]$ for any position *pos*. Of course, we need to seek the exact position on the extracted phrase[1] for the substring decompression problem. Therefore, our method takes $O(N + \ell)$ time in the worst case, but it reduces to $O(N/n + \ell)$ time in the average. Moreover, we need only the auxiliary index structure to find the corresponding phrase, namely, the whole compressed data is not required to load on memory.

We implemented our method with exploiting Re-Pair-VF, and carried out several experiments to see the performance. The results show that our method solves the substring decompression problem much faster than FOLCA in practice.

[1] Note that each codeword corresponds to a phrase.

6.2.1 Rank/Select Dictionary

To access directly to a compressed sequence effectively, the index must be represented by the appropriate data structure. For this problem, we focused on Rank/Select dictionaries. A Rank/Select dictionary is a succinct data structure for a bit string B supporting the following queries:

$$\begin{aligned}
&\mathrm{Rank}(B, i, k) := |\{n \in [0 \dots i] \colon B[i] = k\}| \text{ and} \\
&\mathrm{Select}(B, i, k) := \min\{n \in B \colon \mathrm{Rank}(B, n, k) = i\}, \\
&\text{for } k \in \{0, 1\}.
\end{aligned}$$

Therefore, $\mathrm{Rank}(B, i, k)$ is number of k's in $B[0..i]$ and $\mathrm{Select}(B, i, k)$ is ith appearance position of k in B. Hereafter, $\mathrm{Rank}(B, i, k)$ and $\mathrm{Select}(B, i, k)$ are referred to as $\mathrm{Rank}_k(B, i)$ and $\mathrm{Select}_k(B, i)$ respectively. Further, in case the bit string is self-evident from the context, $\mathrm{Rank}(B, i, k)$ and $\mathrm{Select}(B, i, k)$ are referred to as $\mathrm{Rank}_k(i)$ and $\mathrm{Select}_k(i)$ respectively.

There are number of studies aiming to reduce the storage for a bit string to theoretic lower bound without loss of fast Rank/Select operation. Especially among them, there are three major implementation, DARRAY [30], SDARRAY [30], and RRR [31]. DARRAY is suitable for dense bit string. On the other hand, when the density of a bit string is sparse, SDARRAY is suitable. RRR is suited for the bit strings where the appearances of 1 bits and 0 bits are biased, for instance, the length of it is $2n$ and all first n bits of it are 0 and all last n bits are 1. Either Rank or Select operation is done in constant time with DARRAY and RRR. As for SDARRAY, Select operation is performed in constant time. Meanwhile, Rank requires $O(\log(n/m))$ time where n is the length of the bit string and m is the number of 1's in it. However, the average time complexity of Rank is constant and practically fast.

6.2.2 Fast Direct Access Method on Re-Pair-VF compressed Text

We propose a fast direct access method over Re-Pair-VF compressed text here. First, the index structure is generated as follows:

1. Compress the input text by Re-Pair VF algorithm and obtain dictionary R and compressed sequence σ.

2. Generate the bit sequence B with length N as follows:

$$B[i] = \begin{cases} 1 & \text{if } T[i] \text{ is the last character of a phrase,} \\ 0 & \text{otherwise.} \end{cases} \tag{6.1}$$

   Figure 6.4 illustrates this index structure.

3. Construct an index data structure I in $o(N)$ bits for bit sequence B by using succinct data representation technique.

After generating the index structure I, dictionary D, index structure I, and sequence σ are saved into compressed files.

Next, the direct access to the original text over the compressed sequence is performed as follows:

1. Load index structure I and dictionary D from the compressed files.

2. Given the input position *pos* on the original text, the position of the target codeword is determined by $\mathrm{Rank}_1(I, pos)$. ($\mathrm{Rank}_1(i)$ returns the number of 1's appear up to the position i on the bit sequence, therefore, we obtain the number of codewords up to the position *pos* by performing this operation.)

3. Load the codeword $\sigma[\mathrm{Rank}_1(pos)]$ from the compressed file.

4. Decompress $\sigma[\mathrm{Rank}_1(pos)]$ using dictionary D, and then, original text from $T[pos]$ is obtained.

We assume that each codeword length is fixed because the text is encoded by VF coding. This assumption enables skipping to the target codeword in constant time. Thus, our proposed direct access method runs in $O(pos)$ time.

6.3 Experiments

6.3.1 Evaluation of ADS method and Re-Use algorithm

We conducted computational experiments using several large datasets. The experiments were performed on a workstation equipped with a 3.6 GHz Intel Xeon (R) processor with 32 GB RAM, operating on Ubuntu 12.04. In the experiments we utilized *Pizza & Chili corpus*[2], which contains an English text (*english*), an XML text (*dblp.xml*), a DNA sequence (*dna*), and a protein sequence (*protein*). We generated the following large texts from the datasets as test data.

- *english*: an English language text whose size is 2.2 GB.
- *concat*: a text of 2 GB which consists of 1 GB of *english*, 0.5 GB of *dblp.xml*, and 0.5 GB of *dna*.
- *chaos*: a text of 2 GB which repeats the same block of 512 MB, where each block consists of four 128 MB subblocks of *english*, *dblp.xml*, *dna*, and *protein*.

We implemented ADS method, in addition to Re-Use methods stated in Section 6.1.1, which are First Block Sampling method (1stBlock), Block-Head Sampling method (Head), and Random Sampling method (Random). All of them are written in C language.

[2] *Pizza & Chili corpus* : `http://pizzachili.dcc.uchile.cl/texts.html`

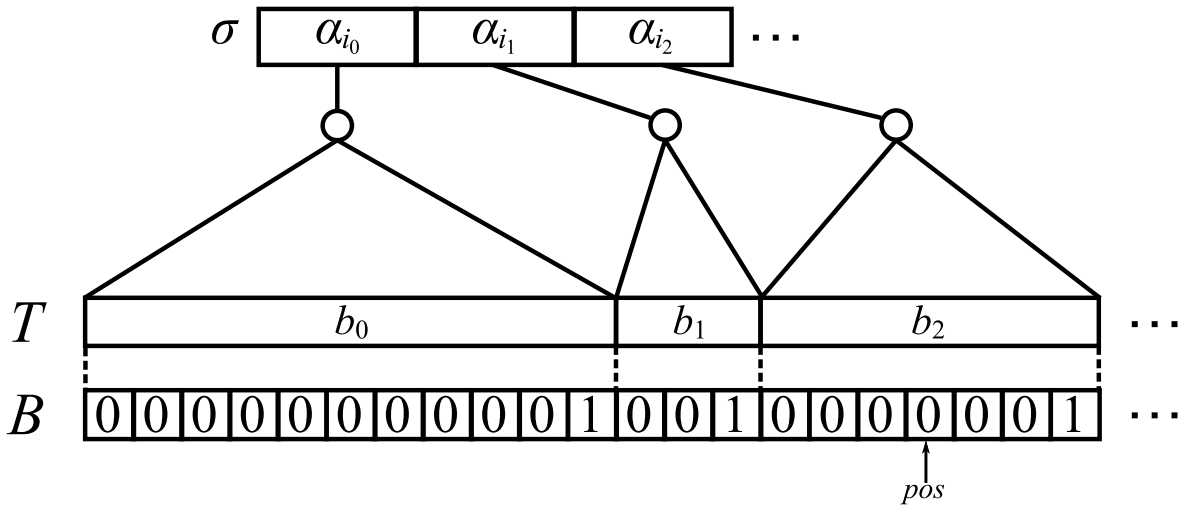

Figure 6.4: The proposed index structure for direct access on compressed text by VF coding. In this figure, $\sigma := \alpha_{i_0}\alpha_{i_1}\alpha_{i_3}\ldots$ is a sequence of VF a coding method. The triangles represents decompression process of each member of the σ. Let T and B be the original text and the bit sequence constructed from σ and T, respectively. The jth phrase in T is denoted by b_j, which is decomposed by the compression algorithm, corresponding to α_{i_j}. Given the input position *pos* on the original text, the position of the target codeword determined by $\mathrm{Rank}_1(B, pos)$. In this figure, we have $\mathrm{Rank}_1(B, pos) = 2$. Therefore, $T[pos]$ lies in the second codeword of σ. And then, decompress the codeword $\sigma[\mathrm{Rank}_1(B, pos)]$, and original text from $T[pos]$ is obtained.

We conducted three experiments. First, we investigated a change of performance of ADS method for the minimum frequency threshold. Next, we compared ADS method to Re-Use methods in the compression ratio and speed for the above three texts. Moreover, we compared ADS method to Re-Merge [42][3], gzip[4], bzip2[5], and lzma[6].

We measured the compression ratio and time by varying minimum frequency threshold m. Figure 6.5 shows the changes of the total compressed data size (the compressed text plus the dictionaries), the compressed text size, and the dictionary size of ADS method for m. From Figure 6.5, we see that the dictionary size becomes smaller when the threshold was set smaller. This suggests that much more rules are shared between consecutive blocks when smaller threshold is used, and thus the total dictionary size becomes smaller. On the other hand, there is no change in compressed text size when the threshold varied. Totally, according to the decreasing of the dictionary size, the whole compression ratio becomes better for a smaller threshold. In this experiment, we obtained the best in compression ratio for $m = 1$. Therefore, we set $m := 1$ for the rest experiments. For the compression speed, the change of m does not affect in any appreciable way. From the above observation, a smaller threshold is better in any cases.

It is necessary to examine the effect when ADS method applies to the text, in which the context varies greatly. In this experiment, we applied ADS and Re-Use algorithms to *concat*, and measured the compression ratio and time for various parameters. This results are shown in Tables 6.1 and 6.2. From Table 6.1, we see that ADS achieved the best compression ratio for *english* and *concat*. From Table 6.2, it is shown that ADS is slightly slower than Re-Use algorithms due to reconstruction of the shared dictionary.

[3] *Re-Store software* : `http://ihome.cuhk.edu.hk/~b126594/ja/restore.html`

[4] gzip : `http://www.gzip.org/`

[5] bzip2 : `http://www.bzip.org/`

[6] lzma : `http://www.7-zip.org/`

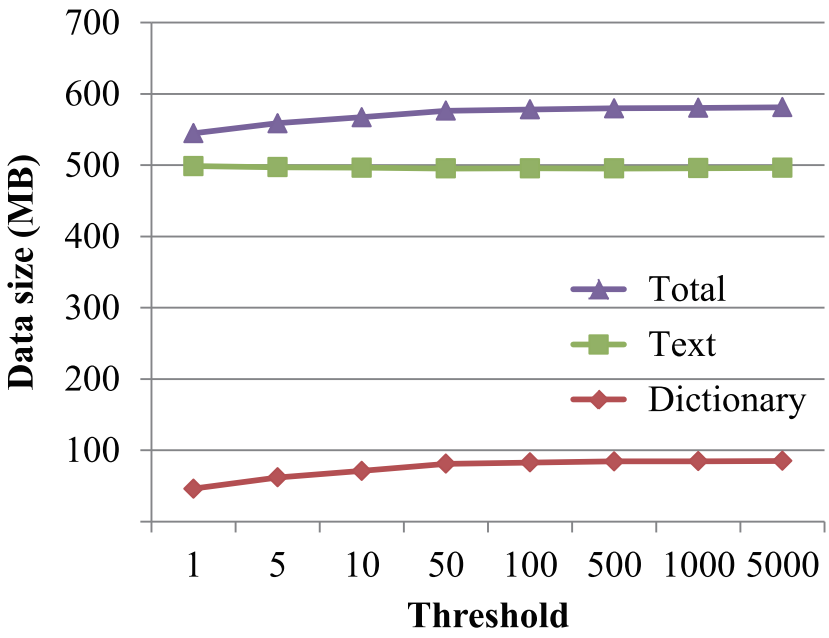

Figure 6.5: Compressed data sizes for ADS on *concat* for various minimum frequency threshold m. In this figure, Total, Text, and Dictionary respectively indicate the total compressed data size, the compressed text size, and the dictionary size. Concerning the parameters, the block size and codeword length are fixed at 128 MB and 20 bits, respectively.

Table 6.1: Compression ratios for ADS and Re-Use algorithms (1stBlock, Head, and Random). In this table, the compression ratios are listed in percent. Concerning the parameters, the block size, codeword length, shared dictionary size (only in Re-Use algorithms), and minimum frequency threshold (only in ADS) are fixed at 128 MB, 20 bits, 37.5% of the total dictionary size, and 1, respectively.

method	*english*	*concat*	*chaos*
Re-Use (1stBlock)	30.84	28.25	36.14
Re-Use (Head)	30.19	28.20	35.47
Re-Use (Random)	30.46	28.14	35.38
ADS	29.84	27.23	35.80

Table 6.2: Compression time for ADS and Re-Use algorithms (1stBlock, Head, and Random). In this table, the compression times are listed in second. Concerning the parameters, the block size, codeword length, shared dictionary size (only in Re-Use algorithms), and minimum frequency threshold (only in ADS) are fixed at 128 MB, 20 bits, 37.5% of the total dictionary size, and 1, respectively.

method	*english*	*concat*	*chaos*
Re-Use (1stBlock)	587.71	488.41	516.53
Re-Use (Head)	620.50	517.48	547.39
Re-Use (Random)	623.70	523.00	554.81
ADS	594.50	494.71	525.27

In order to examine general performance of ADS, we also applied other four compression methods, Re-Merge, gzip, bzip2, and lzma on *concat*. The results obtained using ADS and the other methods are shown in Table 6.3. Gzip, bzip2, and lzma were executed using both the -1 and -9 options. The former option emphasizes compression time, while the latter does compression ratio. Table 6.3 shows that there is no method of which all three criteria (compression ratio, compression time, and decompression time) are superior than ADS. For example, ADS achieved better compression ratio than gzip and it is comparable to bzip2 -9. Lzma -9 and Re-Merge achieved better compression ratio than ADS, however, they took a lot of time. Also we can see that ADS runs slowly compared to other method in compression. On the other hand, decompression is faster than bzip2 and Re-Merge.

6.3.2 Evaluation of Indexing

We conducted computational experiments using several datasets to evaluate the efficiency of our proposed method. The experiments were performed on a workstation equipped with a 3.6 GHz Intel Xeon (R) processor with 32 GB RAM, operating on Debian GNU/Linux 6.0.8. In the experiments we utilized *dna* and the first 500MB of *english* and *proteins* from *Pizza & Chili Corpus*[7]. Our proposed methods are denoted by RVF+RRR, RVF+SDA, and RVF+DA, where RVF stands for Re-Pair-VF. RRR, SDA, and DA respectively represent RRR [31], SDARRAY [30], and DARRAY [30], which are methods for storing the index structure B, implemented by Claude[8].

We compared compression ratios, compression times, and substring extraction times of RVF w/o index, RVF+DA, RVF+SDA, RVF+RRR, and FOLCA [22], where RVF w/o index is a method that scans the compressed text that encoded by Re-Pair-VF from the beginning of it to search the given position without additional succinct in-

[7] `http://pizzachili.dcc.uchile.cl/texts.html`

[8] `https://code.google.com/p/libcds/`

dexable data structure. We chose 0.5 as load factor for FOLCA. Compression times are calculated by the sum of compression time for Re-Pair-VF and index construction time for the proposed methods. They are shown in Table 6.4. The results show that FOLCA is the fastest of all the methods we compared and that RVF+RRR is the fastest and RVF+SDA is the slowest of the proposed methods. The compression ratios are calculated by (compressed file size + index size)/(original file size) for the proposed methods, and by (compressed file size)/(original file size) for RVF w/o index and FOLCA. They are shown in Table 6.5. The compression ratios of the proposed methods are worse than those of RVF w/o index due to the additional indexes. The results show that FOLCA is the best for dna and proteins.500MB, and that RVF+RRR is the best for english.500MB. The compression ratios of RVF+RRR is the best of proposed method. Although those of RVF+SDA is the worst for dna, those of RVF+DA is the worst for english and proteins. The reason is that the density of bit sequence B for dna, which is 0.28, is higher than those for proteins.500MB and english.500MB, which are respectively 0.13 and 0.11.

Next, we show the results of substring extraction. We measured the times required for substring extraction beginning position 0–400000000 with length 10. The experimental results are shown in Figures 6.6 – 6.8. They show that RVF w/o index runs linear to the beginning position and that FOLCA and our proposed methods run constant to it. They also show that all of our proposed methods are faster than FOLCA. The reason is that our proposed methods do not require to read the whole compressed data while FOLCA requires to do the entire one. They load rules and the bit sequence B at first and then determine which codeword does contain the character at given position with Rank and Select operation on B. It should be noted that we utilize variable-to-fixed length codes, i.e., all the codewords have fixed length. Therefore the codeword can be obtained in $O(1)$ time by jumping codewords before it. The results of RVF+DA and RVF+SDA are almost the same. However RVF+RRR is 2–8 times

Table 6.3: Compression results for *concat* using ADS, Re-Merge, gzip, bzip2, and lzma. Compression ratios are given in percent, and times are given in second.

method	comp. ratio (%)	comp. time (sec)	decomp. time (sec)
ADS	27.23	494.71	33.80
Re-Merge	22.66	7907.85	51.18
gzip -1	38.91	33.22	16.47
gzip -9	32.42	360.55	13.55
bzip2 -1	28.76	156.23	60.07
bzip2 -9	25.18	169.23	65.27
lzma -1	34.68	134.20	42.81
lzma -9	21.56	1994.18	23.32

Table 6.4: Compression times for index in seconds.

Method	dna	english.500MB	proteins.500MB
RVF w/o index	98.794	151.740	169.339
FOLCA	72.798	116.101	114.823
RVF+DA	106.262	161.824	180.002
RVF+SDA	109.864	163.493	181.687
RVF+RRR	103.541	159.260	177.435

slower than those. The reason is creating Rank/Select dictionary with RRR requires long time for constructing additional data structures.

6.4 Chapter Summary

In this chapter, we discussed methods for applying VF coding to large texts and their efficiency. We propose large text compression methods for VF coding by dividing the input text into blocks and a direct accessing method on a compressed text by VF coding. We additionally give experimental results of each proposed method in the chapter.

We proposed Re-Use method and ADS method that divide the input text into fixed length blocks and encodes each block using the Re-pair algorithm. Re-Use shares a part of the dictionary among all blocks and all blocks are encoded using codewords of the same length. ADS method adaptively reconstructs the shared dictionary for block-wise compression by using Re-Pair. We also empirically showed that Re-Use and ADS method runs much faster than Re-Merge [42], and obtain similar compression ratios to bzip2 for several combinations of the parameters. Although they have no outstanding feature for both compression ratio and speed, none of state-of-the-art compression tools covers for all aspects. A complicated variable-length encoding can obtain much better compression ratio with an additional sacrifice of compression speed. Conversely speaking, our method achieves a comparable compression ratio to bzip2 or lzma by using fixed-length codes, which is preferable to handle the compressed text.

We also proposed a method that solves the substring decompression problem fast in practice by adding an index structure to Re-Pair-VF code in this chapter. The experimental results showed that a compression ratio of proposed method was better than that of FOLCA on English text and that substring decompression of proposed method is faster than FOLCA.

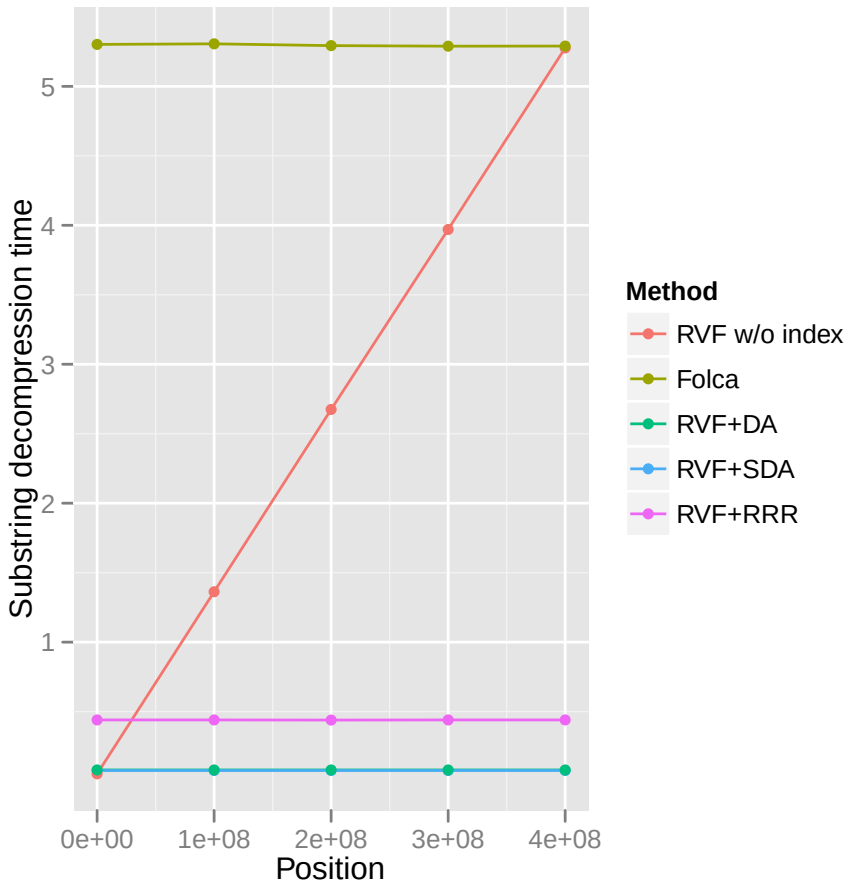

Figure 6.6: Substring extraction times for english.500MB.

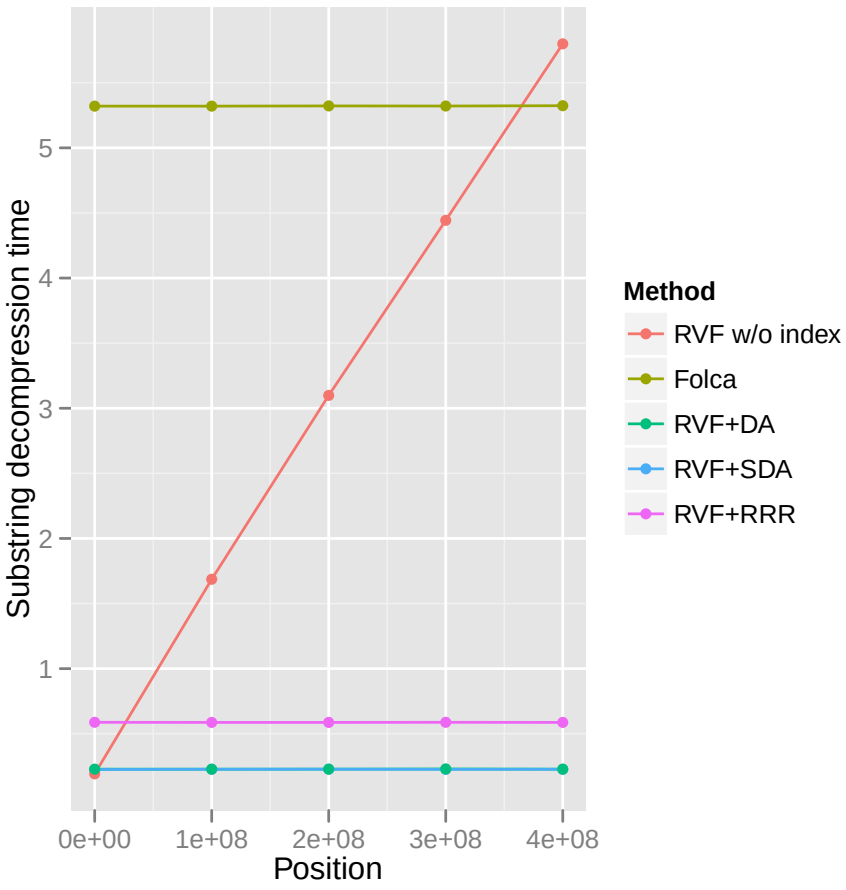

Figure 6.7: Substring extraction times for proteins.500MB.

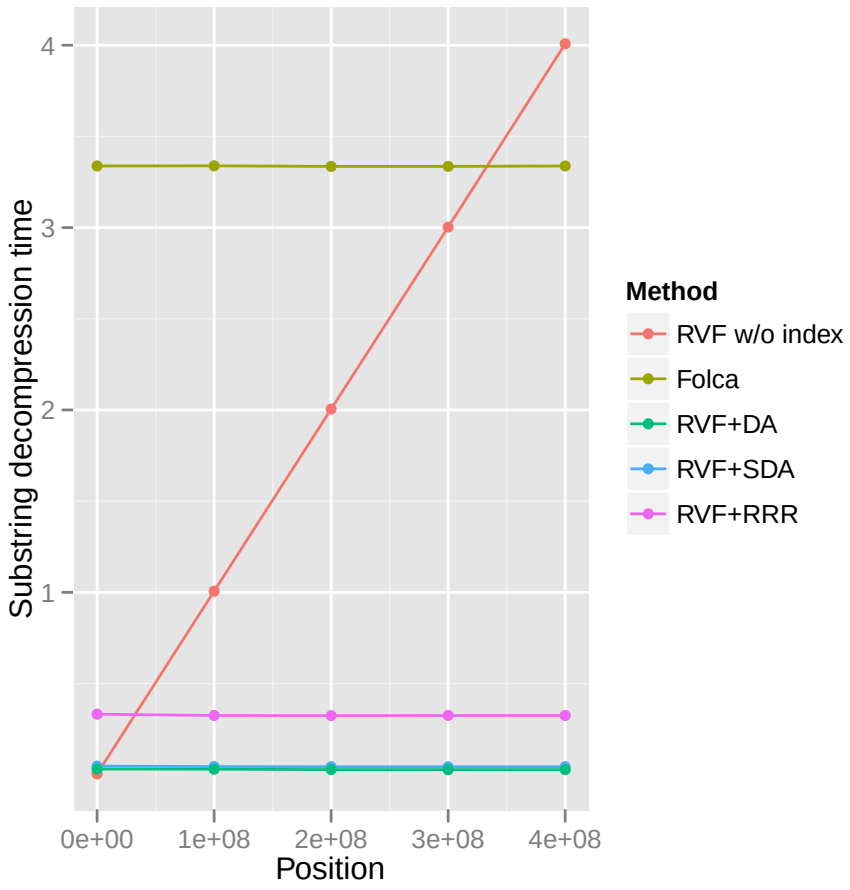

Figure 6.8: Substring extraction times for dna.

Table 6.5: Compression ratios in percentage.

Method	dna	english.500MB	proteins.500MB
RVF w/o index	28.43	30.53	46.27
FOLCA	40.69	47.25	50.34
RVF+DA	47.83	47.62	63.78
RVF+SDA	60.86	45.54	62.39
RVF+RRR	41.05	38.60	54.17

Chapter 7

Concluding Remarks

7.1 Summary of the Results

In this thesis, we addressed the problem of improving variable-length-to-fixed-length coding (VF coding) algorithms. We proposed several coding algorithms and experimentally showed their performance.

In Chapter 3, we presented an efficient algorithm for AIVF coding, which integrates the multiple parse trees of an AIVF coding into one, called a VMA tree, and simulates the encoding process and the decoding process on it. We also estimated the number of nodes in the VMA tree. Roughly translated, it is larger than $M \ln(|\Sigma| + 1)$ and bounded by $M|\Sigma|$, where M and $|\Sigma|$ are the number of codewords and the alphabet size, respectively. Moreover, we conducted several experiments to evaluate the compression performance of three compression methods: Tunstall coding, AIVF coding, and VMA coding. The compression ratios of VMA/AIVF coding methods are better than those of Tunstall coding by 20% on natural language texts. Although the decompression of VMA coding is slower than that of AIVF, the compression of VMA coding is up to six times faster than that of AIVF coding.

In Chapter 4, we presented a method of improving VF coding by training the parse

tree. The experimental results showed that our method applying to the STVF coding proposed by Kida [13] reached to the level of state of the art compression methods in compression ratio in fact. Moreover, Tunstall coding with training runs about twice faster than BPEX, which is an excellent VF coding method proposed by Maruyama *et al.* [23], when we gain almost the same compression ratios. VF coding with training are stable and widely applicable to various data: not only English language texts, but also Japanese texts, DNA data, and so on.

In Chapter 5, we proposed a new VF coding method that utilizes the Re-Pair algorithm [21], which we named as Re-Pair-VF. Although the Re-Pair-VF algorithm uses fixed-length codewords, it delivered extremely good compression performance. In fact, the experimental results demonstrated that the proposed method was much superior to the existing VF coding methods, including the method we proposed at the previous chapter, in terms of the compression ratio and compression time. We also showed that pattern matching on compressed texts compressed by Re-Pair-VF could be performed much faster than an ordinary decompress-then-search approach.

In Chapter 6, we discussed methods for applying VF coding to large text compression. We proposed two methods for VF coding that divides the input text into blocks and then compresses each block. These techniques reduces memory usage for a large text input, but deteriorates the compression ratio. Our method controls the depression in compression ratio by sharing a part of dictionaries for all blocks. We experimentally showed that compression ratio of our method reached to the level of standard well-known compression tools such as gzip and bzip2. Moreover, we discussed in the chapter a direct accessing method on a compressed text by VF coding. It is a method that solves the substring decompression problem fast in practice by adding a succinct index structure to the Re-Pair-VF coding. The experimental results showed that the compression ratio of our method was better than that of FOLCA, which is an excellent online grammar compression method with variable length codewords proposed by

Maruyama [22], and also showed that the substring decompression of our method runs in almost ten times faster than FOLCA.

Through this study, the author succeeded to develop coding algorithms that are accord with ease of processing on the compressed data, while keeping comparable performance in compression ratio and decompression speed with the state-of-the art non-VF coding algorithms.

7.2 Future Researches

Compression methods we gave in this thesis considerably improved compression ratio, compression speed, and decompression speed of VF coding. Their compression ratio and decompression speed exceed to those of well-known compression tools, and their compression speed reaches to the top of the existent compression methods. However, compared to the fastest one, our methods are opened a lead. Therefore, to develop a much faster VF coding algorithm is one of our future works.

We showed that dividing the input text into blocks enabled us to compress a large text with VF coding in less memory. This method still requires offline processing for each block. It is not sufficient for processing stream data. If processing a block takes time more than the time that the stream comes to a system, a buffer for the next block could overflow. Therefore, our future work includes to develop an online VF coding algorithm that works in real time.

Massive stream data are usually discarded after extracting significant information. In such case, we have to determine what information is to be extracted in advance. That is, under present circumstances, we can not perform any data analysis which is newly needed by using all the past data. We proposed a fast method for accessing arbitrary position on compressed data in Chapter 6. We assumed there that the whole dictionary used during compression was stored in the main memory. However, the

dictionary can not always be stored in the main memory when operating a huge amount of compressed data. Therefore, our another future work is to devise a fast accessing method on compressed data with consideration of storage device hierarchy.

Bibliography

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In James A. Storer and Martin Cohn, editors, *Proceedings, Data Compression Conference, March 24–27, 1992, Snowbird, Utah*, pages 279–288. IEEE Computer Society, 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[3] P. Bille, G. Landau, R. Raman, K. Sadakane, S. Satti, and O. Weimann. Random access to grammar-compressed strings. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–389. Society for Industrial and Applied Mathematics, 2011.

[4] N. Brisaboa, A. Fariña, J.-R. López, G. Navarro, and E. Lopez. A new searchable variable-to-variable compressor. In James A. Storer and Michael W. Marcellin, editors, *Proceedings, Data Compression Conference, 24–26 March 2010, Snowbird, Utah*, pages 199–208. IEEE Computer Society, 2010.

[5] N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In Mario A. Nascimento, Edleno S. de Moura, and Arlindo L. Oliveira, editors, *String Pro-*

cessing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 2003, Proceedings, volume 2857 of *Lecture Notes in Computer Science*, pages 122–136. Springer-Verlag, 2003.

[6] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Dynamic lightweight text compression. *ACM Transactions on Information Systems*, 28:10:1–10:32, July 2010.

[7] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In Fabrisio Sebastiani, editor, *Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 2003, Proceedings*, volume 2633 of *Lecture Notes in Computer Science*, pages 468–481. Springer-Verlag, 2003.

[8] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[9] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[10] P. Gage. A new algorithm for data compression. *C Users Journal*, 12:23–38, February 1994.

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

[12] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Fast q-gram mining on SLP compressed strings. *Journal of Discrete Algorithms*, 18:89–99, January 2013.

[13] T. Kida. Suffix tree based VF-coding for compressed pattern matching. In James A. Storer and Michael W. Marcellin, editors, *Proceedings, Data Compres-*

sion Conference, 15–19 March 2009, Snowbird, Utah, page 449. IEEE Computer Society, Mar. 2009.

[14] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[15] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In Maxime Crochemore and Mike Paterson, editors, *Combinatorial Pattern Matching, 10th Annual Symposium, CPM99, Warwick University, UK, July 1999, Proceedings*, volume 1645 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1999.

[16] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In James A. Storer and Martin Cohn, editors, *Proceedings, Data Compression Conference, March 30–April 1, 1998, Snowbird, Utah*, pages 103–112. IEEE Computer Society, 1998.

[17] J. Kieffer and E.-H. Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[18] J. Kieffer, E.-H. Yang, G. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.

[19] S. Klein and M. Ben-Nissan. Using Fibonacci compression codes as alternatives to dense codes. In James A. Storer and Michael W. Marcellin, editors, *Proceedings, Data Compression Conference, 25–27 March 2008, Snowbird, Utah*, pages 472–481. IEEE Computer Society, 2008.

[20] S. Klein and D. Shapira. Improved variable-to-fixed length codes. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *String Processing and Infor-*

mation Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 2008, Proceedings, volume 5280 of *Lecture Notes in Computer Science*, pages 39–50. Springer-Verlag, 2008.

[21] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[22] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-online grammar compression. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval, 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 218–229. Springer International Publishing, 2013.

[23] S. Maruyama, Y. Tanaka, H. Sakamoto, and M. Takeda. Context-sensitive grammar transform: Compression and pattern matching. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 2008, Proceedings*, volume 5280 of *Lecture Notes in Computer Science*, pages 27–38. Springer-Verlag, 2008.

[24] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[25] A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

[26] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[27] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In Maxime Crochemore and Mike Paterson, ed-

itors, *Combinatorial Pattern Matching, 10th Annual Symposium, CPM99, Warwick University, UK, July 1999, Proceedings*, volume 1645 of *Lecture Notes in Computer Science*, pages 14–36. Springer-Verlag, 1999.

[28] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In Raffaele Giancarlo and David Sankoff, editors, *Combinatorial Pattern Matching, 11th Annual Symbosium, CPM 2000, Montreal, Canada, June 2000, Proceedings*, volume 1848 of *Lecture Notes in Computer Science*, pages 166–180. Springer-Verlag, 2000.

[29] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In James A. Storer and Martin Cohn, editors, *Proceedings, Data Compression Conference, March 29–31, 1994, Snowbird, Utah*, pages 244–253. IEEE Computer Society, 1994.

[30] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. Society for Industrial and Applied Mathematics, 2007.

[31] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[32] J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In Alberto Apostolico and Masayuki Takeda, editors, *Combinatorial Pattern Matching, 13th Annual Symposium, CPM 2002, Fukuoka, Japan, July 2002, Proceedings*, volume 2373 of *Lecture Notes in Computer Science*, pages 42–52. Springer-Verlag, 2002.

[33] D. Salomon and G. Motta. *Handbook of data compression.* Springer-Verlag, 5th edition, 2010.

[34] K. Sayood. *Introduction to data compression.* Morgan Kaufmann, 4th edition, 2012.

[35] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In Giancarlo Bongiovanni, Giorgio Gambosi, and Rossella Petreschi, editors, *Algorithms and Complexity, 4th Italian Conference, CIAC 2000, Rome, Italy, March 2000, Proceedings*, volume 1767 of *Lecture Notes in Computer Science*, pages 306–315. Springer-Verlag, 2000.

[36] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In Raffaele Giancarlo and David Sankoff, editors, *Combinatorial Pattern Matching, 11th Annual Symbosium, CPM 2000, Montreal, Canada, June 2000, Proceedings*, volume 1848 of *Lecture Notes in Computer Science*, pages 181–194. Springer-Verlag, 2000.

[37] 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.

[38] B. Tunstall. *Synthesis of noiseless compression codes.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1967.

[39] A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622 –628, 2000.

[40] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[41] J. Verhoeff. A new data compression technique. *Annals of Systems Research*, 6:139–148, 1978.

[42] R. Wan and A. Moffat. Block merging for off-line compression. *Journal of American Society for Information Science and Technology*, 58(1):3–14, January 2007.

[43] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching&Automata Theory, October 15–17, 1973*, pages 1–11. IEEE Computer Society, 1973.

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

[45] S. Yoshida, T. Uemura, T. Kida, T. Asai, and S. Okamoto. Improving parse trees for efficient variable-to-fixed length codes. *Journal of Information Processing*, 20(1):238–249, 2012.

[46] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[47] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.