View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

A Space-Saving Approximation Algorithm for Grammar-Based Compression

Hiroshi Sakamoto
Kyushu Institute of Technology
hiroshi@ai.kyutech.ac.jp

Abstract

A space-efficient approximation algorithm for the grammar-based compression problem, which requests for a given string to find a smallest context-free grammar deriving the string, is presented. For the input length n and an optimum CFG size g, the algorithm consumes only $O(g \log g)$ space and $O(n \log^* n)$ time to achieve $O((\log^* n) \log n)$ approximation ratio to the optimum compression, where $\log^* n$ is the maximum number of logarithms satisfying $\log \log \cdots \log n > 1$. This ratio is thus regarded to almost $O(\log n)$, which is the currently best approximation ratio. While g depends on the string, it is known that $g = \Omega(\log n)$ and $g = O\left(\frac{n}{log_k n}\right)$ for strings from k-letter alphabet [12].

1 Introduction

The grammar-based compression problem is to find a smallest context-free grammar generating just single string. Such a CFG requires that every nonterminal is derived from only one production rule, say, deterministic. The problem deeply relates to factoring problems for strings, and the complexity of similar minimization problems have been rigorously studied. For example, Storer [20] introduced a factorization for a given string and showed the problem is NP-hard. De Agostino and Storer [2] defined several online variants and proved that those are also NP-hard.

As non-approximability results, Lehman and Shelat [13] showed that the problem is APX-hard, i.e. it is hard to approximate this problem within a constant factor (see [1] for definitions). They also mentioned its interesting connection to the *semi-numerical problem* [9], which is an algebraic problem of minimizing the number of different multiplications to compute the given integers and has no known polynomial-time approximation algorithm achieving a ratio $o(\log n / \log \log n)$. Since the problem is a special case of the grammar-based compression, an approximation better than this ratio seems to be also hard.

On the other hand, various practical algorithms for the grammar-based compression have been devised so far. LZW [21] including LZ78 [24], and BISECTION [8] are considered as algorithms that computes straight-line programs, CFGs formed from Chomsky normal form formulas. Also algorithms for restricted CFGs have been presented in [6, 10, 15, 16, 22]. Lehman and Shelat [13] proved the upper bounds of the approximation ratio of these practical

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1693

algorithms, as well as the lower bounds with the worst-case instances. For example, BISECTION algorithm achieves an approximation ratio no more than $O((n/\log n)^{1/2})$. All those ratios, including the lower-bounds, are larger than $O(\log n)$.

Recently polynomial-time approximation algorithms for the grammar-based compression problem have been widely studied and the worst-case approximation ratio has been improved. The first $\log n$-approximation algorithm was developed by Charikar et al. [4]. Their algorithm guarantees the ratio $O(\log(n/g))$, where g is the size of a minimum deterministic CFG for an input. Independently, Rytter presented in [17] another $O(\log(n/g))$-approximation algorithm that employs a suffix tree and the LZ-factorization technique for strings. Sakamoto also proposed in [19] a simple linear-time algorithm based on Re-pair [10] and achieving ratio $O(\log n)$; Now this ratio has been improved to $O(\log(n/g))$.

The ratio $O(\log(n/g))$ achieved by these new algorithms is theoretically sufficiently small. However, all these algorithms require $O(n)$ space, and it prevents us to apply the algorithms to huge texts, which is crucial to obtain a good compression ratio in practice. For example, the algorithm Re-pair [10] spends $5n + n^{1/2}$ space on unit-cost RAM with the input size n.

This state motivates us to develop a sub-linear space $O(\log n)$-approximation algorithm for the grammar-based compression. We presented a simple algorithm [18] that repeats substituting one new nonterminal symbol to all the same and non-overlapping two contiguous symbols occurring in the string. This is carried out by utilizing idea of the lowest common ancestor of balanced binary trees, and no real special data structure, such as suffix tree or occurrence frequency table, is requested. In consequence, the space complexity is nearly equal to the total number of created nonterminal symbols, each of which corresponds to a production rule in Chomsky normal form. This algorithm was applied to *Compressed Pattern Matching* in [14]. In this paper we improve the algorithm and obtain almost $O(\log n)$-approximation ratio preserving the space complexity.

The size of the final dictionary of the rules is proved by the compactness of LZ-factorization [17] and alphabet reduction technique [5]. This technique requires $\log^* n$ times iteration. Here $\log^* n$ denotes the maximum integer j which satisfies $F(j) \leq n$ for

$$F(0) = 1, \quad \text{and} \quad F(j) = 2^{F(j-1)} \quad (j \geq 1).$$

For instance, $F(3) = 2^4 = 16$, $F(4) = 2^{16} = 65536$, and $F(5) = 2^{65536}$. Thus, $\log^* n$ is almost constant even for sufficiently large n. Our algorithm runs in almost $O(n)$ time and $O(g \log g)$ space preserving the worst-case approximation ratio $O((\log^* n) \log n)$. This ratio is almost the currently best approximation. The memory space is devoted to the dictionary that maps a contiguous pair of symbols to a nonterminal. Practically, in randomized model, space complexity can be reduced to $O(g \log g)$ by using a hash table for the dictionary. In the framework of dictionary-based compression, the lower-bound of memory space is usually estimated by the size of a possible smallest dictionary, and thus our algorithm is nearly optimal in space complexity. Compared to other practical dictionary-based compression algorithms, such as LZ78, which achieves the ratio $\Omega(n^{2/3}/\log n)$, the lower-bound of memory space of our algorithm is considered to be sufficiently small.

The remaining part of this paper is organized as follows. In Section 2, we prepare the definitions related to the grammar-based compression. In Section 3, we introduce the notion of lowest common ancestors in a complete binary tree defined by alphabet symbols. Using this notion, our algorithm decides a fixed priority of all pairs appearing in a current string and replaces them according to the priority. More precisely, a pair is called to be *maximal* if its priority is higher than the neighbors'. The aim of the algorithm is to find as many maximal pairs as possible, and this is performed by iterative application of the alphabet reduction. The algorithm is presented in Section 4 and we analyze the approximation ratio and estimate the time/space efficiency compared with related grammar-based compression algorithms. In Section 5, we summarize this study.

2 Notions and Definitions

In this study we suppose a standard RAM model [11] with the *unit-cost measure*, in which the following assumptions are made. Each value is a primitive data item, the memory required by a given variable is equal to the number of entries in the array that it represents, the memory required by a RAM is equal to the total memory required by its variables, and the time required by a RAM is equal to the number of instructions being executed.

We next recall the notions in formal language theory. Given a sufficiently large integer n for the input length, we assume that the size of any symbol is bounded by $O(\log n)$ bits, and a finite set Σ of symbols is called an *alphabet*. The set of all strings over Σ is denoted by Σ^*, and Σ^i denotes the set of all strings of length just i. The length of a string $w \in \Sigma^*$ is denoted by $|w|$, and also for a set S, the notion $|S|$ refers to the size (cardinality) of S. The ith symbol of w is denoted by $w[i]$. For an interval $[i, j]$ with $1 \leq i \leq j \leq |w|$, the occurrence of a substring from $w[i]$ to $w[j]$ is denoted by $w[i, j]$.

A *repetition* is a string x^k for some $x \in \Sigma$ and some positive integer k. A repetition $w[i, j]$ in w of a symbol $x \in \Sigma$ is *maximal* if $w[i-1] \neq x$ and $w[j+1] \neq x$. It is simply referred by x^+ if there is no ambiguity in its interval in w. Intervals $[i, j]$ and $[i', j']$ with $i < i'$ are *overlapping* if $i' \leq j < j'$, and are *independent* if $j < i'$. A *pair* $u \in \Sigma^2$ is a string of length two, and an interval $[i, i+1]$ is a *segment of* u *in* w if $w[i, i+1] = u$.

A *context-free grammar* (*CFG*) is a quadruple $G = (\Sigma, N, P, s)$ of disjoint finite alphabets Σ and N, a finite set $P \subseteq N \times (N \cup \Sigma)^*$ of *production rules*, and the *start symbol* $s \in N$. Symbols in N are called *nonterminals*. A production rule $a \to b_1 \cdot \cdots \cdot b_k$ in P *derives* $\beta \in (\Sigma \cup N)^*$ *from* $\alpha \in (\Sigma \cup N)^*$ by replacing an occurrence of $a \in N$ in α with $b_1 \cdot \cdots \cdot b_k$. In this paper, we assume that any CFG is *deterministic*, that is, for each nonterminal $a \in N$, exactly one production rule from a is in P. Thus, the language $L(G)$ defined by G is a singleton set. We say a CFG G *derives* $w \in \Sigma^*$ if $L(G) = \{w\}$. The *size of* G is the total length of strings in the right hand sides of all production rules, and is denoted by $|G|$. The aim of grammar-based compression is formalized as a combinatorial optimization problem, as follows:

Problem 1 Grammar-Based Compression
Instance: *A string* $w \in \Sigma^*$.

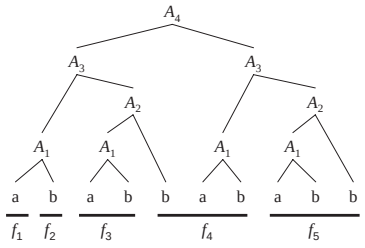

Figure 1: LZ-factorization and CFG derivation.

SOLUTION*: A deterministic CFG G that derives w.*
MEASURE*: The size of G.*

From now on, we assume that every deterministic CFG is in Chomsky normal form, i.e. the size of strings in the right-hand side of production rules is two, and we use $|N|$ for the size of a CFG. Note that for any CFG G, there is an equivalent CFG G' in Chomsky normal form such that $|G'| \leq 2 \cdot |G|$.

The *approximation ratio* of a grammar-based compression algorithm A is defined by the quantity

$$\max_{w \in \Sigma^*} \left\{ \frac{|G_A(w)|}{|G_{opt}(w)|} \right\},$$

where $G_A(w)$ is the CFG computed by A and $G_{opt}(w)$ is an optimum CFG for a string w.

It is known that there is an important relation between a deterministic CFG and a factorization called *LZ-factorization*. The factorization for w, denoted by $LZ(w)$, is the decomposition of w into $f_1 \cdot \cdots \cdot f_k$, where $f_1 = w[1]$, and for each $1 < \ell \leq k$, f_ℓ is the longest prefix of the suffix $w[|f_1 \cdots f_{\ell-1}| + 1, |w|]$ that appears in $f_1 \cdots f_{\ell-1}$, where $f_{\ell-1}$ is empty if $\ell = 1$. Each f_ℓ is called a *factor*. The size $|LZ(w)|$ of $LZ(w)$ is the number of its factors. The following result is used in the analysis of the approximation ratio of our algorithm.

Example 1 The relation of the size of LZ-factorization and CFG is illustrated in Fig. 1. For a string "*ababbabab*", the first two factors are $f_1 = a$ and $f_2 = b$. Similarly, we obtain the sequence

$$f_1 = a,\ f_2 = b,\ f_3 = ab,\ f_4 = bab,\ f_5 = abb.$$

Fig. 1 shows that the size of LZ-factorization is always smaller than or equal to that of any CFG. Note that the size of CFG is defined by $2|N|$.

Theorem 1 ([17]) For any string w and its deterministic CFG G, the inequality $|LZ(w)| \leq |G|$ holds.

This theorem shows that the number of LZ factors is smaller than the size of a minimum CFG for any string.

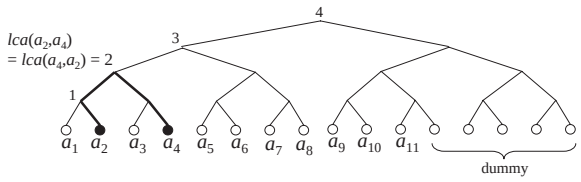

Figure 2: The alphabet tree for $\Sigma \cup N = \{a_1, \ldots, a_{11}\}$.

3 Compression by the Alphabetical Order

In this section we describe the central idea of our grammar-based compression utilizing information only available from individual symbols. The aim is to minimize the number of different nonterminals generated by our algorithm.

A *replacement* $[i, i+1] \to a$ *for* w is an operation that replaces a pair $w[i, i+1]$ with a nonterminal $a \in N$. A set R of replacements is, by assuming some order on R, regarded as an operation that performs a series of replacements to w. In the following we introduce a definition of a set of replacements whose effect on a string is independent of the order.

Definition 1 A set R of replacements for w is *appropriate* if it satisfies the following: (1) At most one of two overlapping segments $[i, i+1]$ and $[i+1, i+2]$ is replaced by replacements in R, (2) At least one of three overlapping segments $[i, i+1]$, $[i+1, i+2]$ and $[i+2, i+3]$ is replaced by replacements in R, and (3) For any pair of replacements $[i, i+1] \to a$ and $[j, j+1] \to b$ in R, $a = b$ if and only if $w[i, i+1] = w[j, j+1]$.

Clearly, for any string w, an appropriate replacement R for w generates the string w' uniquely. In such a case, we say that R *generates* w' *from* w, and write $w' = R(w)$. Intuitively, $w' = R(w)$ is a resulting string by an execution of single loop of our compression algorithm, which continues the process till $|w'| < |w|$.

Our first problem is to find small appropriate replacements, and here we explain the strategies for making pairs in our algorithm.

Alphabet tree: Let d be a positive integer, and let k be $\lceil \log_2 d \rceil$. An alphabet tree T_d for $\Sigma \cup N = \{a_1, \ldots, a_d\}$ is the rooted, ordered complete binary tree whose leaves are labeled with $1, \ldots, 2^k$ from left to right. The *height* of an internal node refers to the number of edges of a path from the node to a descendant leaf. Let h be the height of the lowest common ancestor of leaves i and j. Then we define $lca(a_i, a_j)_d = h$. Usually we omit the index d, and for the simplicity we assume that $lca(i, j)$ is identical to $lca(a_i, a_j)$. Moreover 'log' denotes the binary logarithm throughout this paper.

Example 2 If $|\Sigma \cup N| = 11$, the corresponding alphabet tree and the value of $lca(i, j)$ are illustrated in Fig. 2.

For every string $w \in \Sigma^+$, any maximal repetition $w[i, j] = x^k$ is called *type 1 metablock* and any other occurrence of substring is called *type 2 metablock* of w. For example we illustrate the following factorization by type 1 and 2 metablock:

$$w = abcabbcaaabab = abca \cdot bb \cdot c \cdot aaa \cdot bab$$

Any type 1 metablock α can be compressed to a sufficiently short string. For instance, if $\alpha = b^{2k}$ for a symbol b, α is compressed to A^k by $A \to bb$, and if $\alpha = b^{2k+1}$, α is compressed to A^kB by $A \to bb$ and $B \to b$. The trivial production rule $B \to b$ is produced to replace all symbols in the current string. This strategy is important to achieve our space-saving compression. In the next section, we introduce the general case of such compression called *typical compression*.

For type 2 metablocks, we introduced our iterative compression technique by *lca* and alphabet reduction.

Definition 2 Let w be a type 2 metablock. $w[i, i+1]$ is called to be *maximal* if $lca(w[i], w[i+1]) > lca(w[i-1], w[i]), lca(w[i+1], w[i+2])$, where $w[1, 2]$ is maximal if $lca(w[1], w[2]) > lca(w[2], w[3])$, and the case $w[|w|-1, |w|]$ is similarly defined.

Our idea is to replace all occurrences of maximal pairs prior to others. Any two occurrences of maximal pairs are not overlapping, that is, if $w[i, i+1]$ is maximal, then neither $w[i-1, i]$ nor $w[i+1, i+2]$ is maximal. Thus, we can replace all the occurrences of maximal pairs by appropriate nonterminals. However there is a long substring $w[i, j]$ containing no maximal pair such that $|w[i, j]| = \lceil \log |\Sigma| \rceil$ in worst case. For instance, $a_1a_2a_4 \cdots a_{2^k}$ is one of such strings. For improving such a bound, we compute $lca(w[i], w[i+1])$ iteratively by the following strategy, which is a variant of alphabet reduction [5] defined on integers. We expand this notion to alphabet trees for our compression problem.

Alphabet reduction: Let w be a type 2 metablock. In case $k = 2, \ldots, |w|$ and $w[k-1, k] = a_ia_j$, we define $label(w[k]) = 2 \cdot lca(i, j)$ if $i < j$ and $2 \cdot lca(i, j) + 1$ otherwise. In case $k = 1$ and $w[1, 2] = a_ia_j$, we define $label(w[1]) = 2 \cdot lca(i, j)$ if $i > j$ and $2 \cdot lca(i, j) + 1$ otherwise.

Lemma 1 For each k, if $w[k] \neq w[k+1]$, then $label(w[k]) \neq label(w[k+1])$.

proof. We show that $label(w[\ell+1]) \neq label(w[\ell+2])$ for $w[\ell, \ell+2] = a_ia_ja_k$. In case $(j > i, k)$ or $(j < i, k)$, exactly one of $label(w[\ell+1])$ and $label(w[\ell+2])$ is odd. In case $i < j < k$, we obtain $lca(i, j) \neq lca(j, k)$. Moreover, $label(w[\ell+1]) = 2 \cdot lca(i, j)$ and $label(w[\ell+2]) = 2 \cdot lca(j, k)$ derives $label(w[\ell+1]) \neq label(w[\ell+2])$. The case of $i > j > k$ is similar. *Q.E.D.*

From a string w of length n, a sequence $w' = label(w[1])label(w[2]) \cdots label(w[n])$ is computed. By regarding each integer $\ell = label(w[k])$ as a next alphabet symbol a_ℓ, we then continue the alphabet reduction for the string w iteratively. The purpose of the alphabet reduction is to reduce all symbols to constant integers preserving the structures of substrings. The next lemma shows that the number of iteration is very small.

Lemma 2 After at most $\log^* n$ iterations of alphabet reduction, the label size is 6.

proof. Let $w[k-1, k] = a_ia_j$ and $2 \leq k \leq n = |w|$. The size of the next label of $w[k]$ is reduced to $label(w[k]) \leq \max\{2\lceil \log j \rceil, 2\lceil \log i \rceil\} + 1$ by single iteration. Thus, the alphabet reduction terminates within $\log^* n$ iterations. Moreover, at each iteration, the alphabet size goes from $|\Sigma|$ to at most $2\lceil |\log \Sigma| \rceil$. If $|\Sigma| > 6$, then $2 \log \lceil |\Sigma| \rceil < |\Sigma|$, that is, the next label size is smaller than the current label size. Thus, the final labels are bounded by 6. *Q.E.D.*

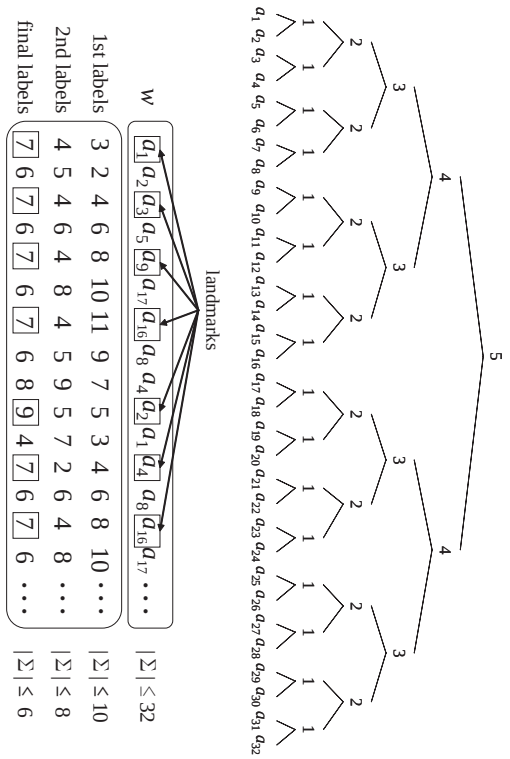

Figure 3: A worst case $\log^*|\Sigma|$-iteration of alphabet reduction and resulting landmarks: each internal node in the tree denotes the value of *lca* for the corresponding leaves.

If different symbols in w are less than or equal to 6, the iteration of alphabet reduction terminates. When the iteration terminates for the string w, the resulting sequences $label(w[1])label(w[2])\cdots label(w[n])$ is called a *final labels*, and a symbol $w[k]$ is called a *landmark* if $label(w[k])$ is maximal, i.e. $label(w[k]) > label(w[k-1]), label(w[k+1])$, where $w[1]$ is maximal if $label(w[1]) > label(w[2])$, and the case $w[|w|]$ is similar.

Here we note that any $w[i,j]$ in type 2 string longer than 6 must contain at least one landmark. Using this property, the aim of our algorithm is to synchronize the landmarks in all occurrences of a same substring.

Example 3 *We show a worst case iteration of alphabet reduction in Fig. 3. In case that* $|\Sigma| \leq 32$*, if* w *is formed by the string presented in Fig. 3,* $\log^*|\Sigma| = 3$ *times iteration is necessary in worst case to obtain the finial label sequence.*

4 Algorithm and Analysis

In this section we introduce an approximation algorithm for the grammar-based compression problem and analyze its approximation ratio to the optimum as well as its space efficiency.

```
1  Algorithm LCA*(w)
2  initialize ℓ = 1 for counter of ℓth loop;
3  factorize w = w_1 w_2 ··· w_m by type 1 and 2 metablock;
4  for each type 1 metablock w_i,
5      compute a typical compression;
6  for each type 2 metablock w_i
7      compute its landmarks w_i[x], w_i[y], ..., w_i[z];
8  replace all pairs w_i[x-1, x], w_i[y-1, y], ..., w_i[z-1, z]
9      by appropriate nonterminals;
10 compute typical compressions for remained substrings in w_i;
11 set ℓth dictionary D_ℓ, ℓ = ℓ + 1, w = w_1 w_2 ··· w_m
12     by the replaced w_is, and goto line 3;
13 repeat this process until all pairs in w are mutually different;
14 output D ∪ {S → w} for D = D_1 ∪ ··· ∪ D_ℓ;
```

Figure 4: The LCA* compression algorithm. A replaced pair $w[i, i+1]$ must be *consistent* with a current dictionary D_ℓ, i.e. $w[i, i+1]$ is replaced by A if a production $A \to BC$ ($BC = w[i, i+1]$) is already registered to a D_ℓ and a new nonterminal is created to replace $w[i, i+1]$ otherwise.

4.1 Algorithm LCA*

Before the description of our algorithm, we first explain a typical compression for a trivial string. The following trivial replacement $R(w) = A_1 \cdots A_k$ is called a *typical compression* for w of length n.

$$A_1 \to w[1,2], A_2 \to w[3,4], \ldots,$$

$$\begin{cases} A_k \to w[n-1, n], \text{ if } n \text{ is even} \\ A_{k-1} \to w[n-2, n-1], A_k \to w[n], \quad \text{otherwise.} \end{cases}$$

The last replacement $A_k \to w[n]$ is called *renaming*. In our compression algorithm, we assume any replacement is *consistent* to a current dictionary D, that is, any replaced pair $w[i, i+1]$ and $w[j, j+1]$ must be replaced by an identical nonterminal if $w[i, i+1] = w[j, j+1]$. The algorithm LCA*(w) is presented in Fig. 4. We describe the outline of LCA*(w) in Fig. 4.

Phase 1 (Line 3):

The algorithm find all type 1 and type 2 metablocks in the input string w. Each metablock w_i is compressed in Phase 2 and 3 individually.

Phase 2 (Line 4 – 5):

Type 1 metablock substring w_i, i.e. a maximal repetition is replaced by a typical compression for w_i. If $|w_i|$ is odd, the last symbol is renamed; This trivial replacement is necessary for our space-saving compression. Such renaming is executed in the next phase.

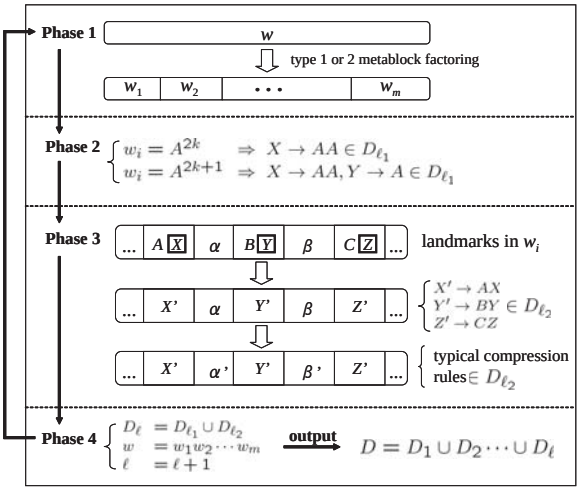

Figure 5: The flow of LCA* for a string w.

Phase 3 (Line 6 – 10):

Type 2 metablock w_i, i.e. $w_i[j] \neq w_i[j+1]$ for all j is replaced. First, all landmarks in w_i are found and for any landmark $w_i[j]$, the pair $w_i[j-1, j]$ is replaced by a nonterminal. Second, if $w_i[k]$ is the nearest landmark from $w_i[j]$, the remained substring $w_i[j+1, k-2]$ is replaced by a typical compression.

Phase 4 (Line 11 – 14):

In ℓth loop, let D_{ℓ_1} and D_{ℓ_2} be the set of production rules produced for type 1 and 2 metablocks, respectively. In this phase, the depth of loop ℓ, the current string w, and the current dictionary D are updated to $\ell = \ell+1$, $w = w_1 w_2 \cdots w_m$ by the compressed metablocks w_i $(1 \leq i \leq m)$, and $D = D_1 \cdots \cup D_\ell$. The above phases are repeated until all symbols are different in a current string. Then, the algorithm outputs the finial dictionary and terminates.

Theorem 2 The running time of LCA*(w) is bounded by $O(n \log^* n)$.

proof. By Lemma 2, the time to compute all landmarks is $O(\log^* n)$ and it is clear that other computation is $O(n)$ time for each loop. Moreover, a current string shrinks in half approximately by single loop of LCA*. Thus, the total length of strings given to LCA* is bounded by $O(n)$. Hence, the time complexity is $O(n \log^* n)$. *Q.E.D.*

4.2 Performance analysis

Before the proof of our approximation ratio, we introduce a notion of occurrences of a substring.

Definition 3 For an occurrence $w[i, j] = \alpha$, we call it a *boundary occurrence* if $w[i-1] \neq w[i]$ and $w[j] \neq w[j+1]$. In case $w[1, j]$, that is, a prefix of w, it is also called a boundary occurrence if $w[j] \neq w[j+1]$, and so is in suffixes.

Lemma 3 Let α be a substring in w satisfying $\alpha = w[\ell, r] = w[\ell', r']$. For any replacement by single loop of LCA*(w), a pair in $w[\ell, r]$ is replaced iff the corresponding pair in $w[\ell', r']$ is replaced by a same nonterminal except at most $\log^* n$ pairs in $w[\ell, r]$ and $w[\ell', r']$.

proof. By Lemma 1 and Lemma 2, for any type 2 metablock, we have following facts:

1. The final labels consist of at most 6 symbols and any label never repeats.
2. The final label of $w[i]$ depends on at most $\log^* n$ symbols to its left.

If $w[\ell, r]$ and $w[\ell', r']$ are both boundary occurrences, there is a unique metablock factorization for them, like $\alpha = \alpha_1 \cdots \alpha_m$. By the above facts, if $|\alpha| > 2\log^* n$, $w[\ell, r]$ and $w[\ell', r']$ contain at least two landmarks in the same positions. Let $w[\ell + i]$ be such the left most landmark and $w[\ell + j]$ be the right most landmark. Thus, by the definition of replacement in our algorithm, the occurrences $w[\ell + i, \ell + j]$ and $w[\ell' + i, \ell' + j]$ are replaced identically.

If $w[\ell, r]$ is not boundary, the shortest boundary occurrence containing $w[\ell, r]$ is formed by $x^+\alpha_1 \cdots \alpha_m y^+$ for $x, y \in \Sigma$ and a metablock α_i. In this case, the replacement of the boundary $\alpha_1 \cdots \alpha_m$ in $w[\ell, r]$ and $w[\ell', r']$ are completely identical since all labels are decided within $\alpha_1 \cdots \alpha_m$. Thus, in this case, disagreement of replacement occurs in the last symbol of x^+ and y^+ only.

Hence, in each case, the replacements of $w[\ell, r]$ and $w[\ell', r']$ for the same substring are identical except at most $\log^* n$ pairs of them. *Q.E.D.*

Theorem 3 The worst-case approximation ratio of the size of a grammar produced by the algorithm LCA* to the size of a minimum grammar is $O((\log^* n) \log n)$.

proof. Let R be the set of appropriate replacements produced by single loop of $LCA^*(w)$. Let g be the size of a minimum CFG for w, and let $w_1 \cdots w_k$ be the LZ-factorization of w. We denote by $\#(w)_R$ the number of different nonterminals produced by R. From the definition of LZ-factorization, any factor w_i occurs in $w_1 \cdots w_{i-1}$, or $|w_i| = 1$.

With lemma 3, any factor w_i and its left-most occurrence are compressed into almost the same strings except $\log^* n$ pairs in them. Thus, by the bound of the number of LZ-factors in Theorem [17], we can obtain the following estimation.

$$\begin{aligned} \#(w)_R &= \#(w_1 \cdots w_{k-1})_R + \log^* n \\ &= \#(w_1 \cdots w_{k-2})_R + 2\log^* n \\ &= O(k \log^* n) \\ &= O(g \log^* n) \end{aligned}$$

This is the number of different nonterminals produced by single loop execution in LCA*(w). Clearly the loop is executed at most $O(\log n)$ times. Therefore, the total number of different nonterminals produced by LCA(w), that is, the size of CFG is $O(g(\log^* n) \log n)$. This derives the approximation ratio. *Q.E.D.*

The memory space required by LCA*(w) can be bounded by the size of data structure to answer the membership query: input is a pair $A_i A_j$; output is an integer k if $A_k \to A_i A_j$ is already created and no otherwise. By Theorem 3,

the size of a current dictionary D_ℓ is bounded by $O(g \log g)$ for each $\ell \geq 1$. Moreover, each symbol A_i in a current string is replaced by a rule of the form $A_j \to A_i$ or $A_j \to YZ$, where $A_i \in \{Y, Z\}$. Thus, $O((g \log g)^2)$-space algorithm is obtained by a naive implementation using look up table. Finally we show that the memory space can be improved to $O(g \log g)$.

4.3 Improving the space efficiency

An idea for improving space complexity of the algorithm is to recycle nonterminals created in the preceding iteration. Let $D(\alpha)$ be the string obtained by applying a dictionary D to a string α. Let D_1 and D_2 be dictionaries such that any symbol in w is replaced by D_1 and any symbol in $D_1(w)$ is replaced by D_2. Then, the decoding of the string $D_2(D_1(w))$ is uniquely determined, even if D_2 reuses nonterminals in D_1 like "$A \to AB$." Thus, we consider that the final dictionary D is composed of $D_1, \ldots, D_\ell$, where D_i is the dictionary constructed in the ith iteration. Since all symbols in w are replaced within a same loop in LCA*, the decoding from the final string w' is uniquely decided by the semantics $D_m(\cdots D_1(w') \cdots) = w$. Such a dictionary is computed by the following function and data structures. Let D_i be the set of productions, N_i the set of alphabet symbols created in the ith iteration, and k_i the cardinality $|N_i|$. We define the function

$$f_i(x, y) = (x - 1)k_i + y \text{ for } x, y = 1, \ldots, k_i.$$

This is a one-to-one mapping from $\{1, \ldots, k_i\} \times \{1, \ldots, k_i\}$ to $\{1, \ldots, k_i^2\}$, and is used to decide an index of a new nonterminal associated to a pair $A_x A_y$, where A_x denotes the xth created nonterminal in N_i.

The next dictionary D_{i+1} is constructed from D_i, N_i, and f_i as follows. In the algorithm LCA*, there are two cases of replacements: one is for replacements of pairs, and the other is renaming. We first explain the case of replacements of pairs. Let a pair $A_x A_y$ in a current string be decided to be replaced. The algorithm LCA* computes the integer $z = f(x, y)$, and looks up a hash table H for z. If $H(z)$ is absent and $N_i = \{A_1, \ldots, A_k\}$, then set $N_i = N_i \cup \{A_{k+1}\}$, $D_i = D_i \cup \{A_{k+1} \to A_x A_y\}$, $H(z) = k + 1$, and replace the pair $A_x A_y$ with A_{k+1}. If $H(z) = k + 1$ is present, then only replace the pair $A_x A_y$ by A_{k+1}. For the case of renaming of a symbol A_x, we can use the nonterminal A_{k+1} such that $z = f_i(x, x)$ and $H(z) = k + 1$. The dictionary D_i constructed in the ith iteration can be divided to D_{i_1} and D_{i_2} such that D_{i_1} is the dictionary for repetitions and $D_{i_2} = D_i \setminus D_{i_1}$. Thus, we can create all productions without collisions, and decode a current string w_{i+1} to the previous string w_i by the manner $D_i(w_{i+1}) \equiv D_{i_1}(D_{i_2}(w_{i+1})) = w_i$.

Theorem 4 The space required by LCA*(w) is $O(g \log g)$ for the size of a minimum CFG for w.

proof. Let $n = |w|$ and ℓ be the number of iterations of loops executed in LCA*(w). By theorem 3, the number $|N_i|$ of new nonterminals created in the ith iteration is $O(g \log^* n)$ for each $i \leq \ell$. To decide the index of a new nonterminal from a pair $A_x A_y$, LCA* computes $z = f_i(x, y)$, $H(z)$, and $k = |N_i|$ for the current N_i. Since $|z| \leq O(\log n)$ and the number of different z is $O(g \log g)$, the space for H is $O(g \log g)$ and $k = O(g \log g)$. Thus, the construction of D_i

Table 1: Performance of compression algorithms: the common lower bound $\Omega(n)$ for the time complexity is omitted, 'α' and '–' denote $\log^* n$ and *unknown*, respectively, and Rytter [17], Welch [21] contains LZ77, LZ78 algorithms by Ziv and Lempel [23, 24], respectively.

Algorithm [*reference*]	Approx. Ratio *upper/lower*	Space *upper/lower*	Time *upper*
proposed	$\alpha \log n$ / –	$g \log g$ / $\log n$	αn
Charikar[4]	$\log n$ / –	n/n	n
Rytter[17]	$\log n$ / –	n/n	n
Sakamoto[19]	$\log n$ / –	n/n	n
Welch[21]	$\left(\frac{n}{\log n}\right)^{\frac{2}{3}}$ / $\frac{\sqrt[3]{n^2}}{\log n}$	n/n	n
Larsson[10]	$\left(\frac{n}{\log n}\right)^{\frac{2}{3}}$ / $\sqrt{\log n}$	n/n	n
Witten[16]	$\left(\frac{n}{\log n}\right)^{\frac{3}{4}}$ / $\sqrt[3]{n}$	n/n	n
Kieffer[7]	– / $\log \log n$	n/n	n
Kieffer[8]	$\left(\frac{n}{\log n}\right)^{\frac{1}{2}}$ / $\frac{\sqrt{n}}{\log n}$	$\sqrt{n \log n}$ / $\sqrt{n}$	n
Apostolico[3]	$\left(\frac{n}{\log n}\right)^{\frac{2}{3}}$ / 1.37	$n \log n/n$	$n \log^2 n$

requires only $O(g \log g)$ space. We can release whole the memory space for D_i in the next loop. Hence, the total space for constructing D is also $O(g \log g)$. *Q.E.D.*

4.4 Comparison with related works

In Table 1, we summarize all the results obtained in this study together with related works on grammar-based compression, where the trivial time complexity $\Omega(n)$ is omitted. In this table, all results concerned with approximation ratio were proved in [12] as well as the upper/lower bounds of the grammar size by each algorithm, which directly derives the complexity bounds. In particular, the reason for $\Omega(n)$ space of almost algorithms is due to the data structures of indexing for substrings. Since our algorithm does not require no index for substring, the required space depends on the hash table, that is, the grammar size only.

5 Conclusion

We presented a space-efficient near linear-time algorithm for the smallest CFG problem. This algorithm guarantees the approximation ratio $O((\log^* n) \log n)$,

which is almost $O(\log n)$ and the memory space $O(g \log g)$ for the minimum CFG size g of input string. This space bound is considered to be sufficiently small since $\Omega(g)$ space is a lower bound for *non-adaptive* dictionary-based compression. In addition, it is known that $\Omega(\log n) \leq g \leq O(\frac{n}{\log_k n})$ [12] for $k = |\Sigma|$. The upper bound of memory space is best in the previously known *polylog*-approximation algorithms. Practically, production rules for renaming occupy comparatively large space in final dictionary. However it is still open whether it is possible to reduce such renaming production rules preserving the time/space complexity.

Acknowledgments

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 18700154, 2006-2007, and Scientific Research (B), 19300008, 2007-2008. The authors would like to thank the anonymous referees of SPIRE2004, ISAAC2006, and IEICE Transactions on Information and Systems for the valuable comments to revise the preliminary version of this paper.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

[2] S. De Agostino, and J.A. Storer. On-Line versus Off-Line Computation in Dynamic Text Compression. *Inform. Process. Lett.*, 59:169–174, 1996.

[3] A. Apostolico, and S. Lonardi. Some Theory and Practice of Greedy Off-Line Textual Substitution. *Proc. Data Compression Conference 1998*, 119-128.

[4] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inform. Theory*, 51(7):2554-2576, 2005.

[5] G. Cormode, and S. Muthukrishnan. The String Edit Distance Matching Problem With Moves. *ACM Trans. Algorithms*, 3(1), 2007.

[6] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage System: a Unifying Framework for Compressed Pattern Matching. *Theor. Comput. Sci.*, 298(1):253-272, 2003.

[7] J. C. Kieffer, and E.-H. Yang. Grammar-Based Codes: a New Class of Universal Lossless Source Codes. *IEEE Trans. Inform. Theory*, 46(3):737–754, 2000.

[8] J. C. Kieffer, E.-H. Yang, G. Nelson, and P. Cosman. Universal Lossless Compression via Multilevel Pattern Matching. *IEEE Trans. Inform. Theory*, IT-46(5), 1227–1245, 2000.

[9] D. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*, Addison-Wesley, 1981.

[10] N.J. Larsson, and A. Moffat. Offline Dictionary-Based Compression. *Proceedings of the IEEE*, 88(11):1722-1732, 2000.

[11] J. van Leeuwen, (Managing Editor). *Handbook of Theoretical Computer Science*, Volume A: Algorithms and Complexity, Elsevier, 1998.

[12] E. Lehman. Approximation Algorithms for Grammar-Based Compression. PhD thesis, MIT, 2002.

[13] E. Lehman, and A. Shelat. Approximation Algorithms for Grammar-Based Compression. *Proc. 20th Ann. ACM-SIAM Sympo. Discrete Algorithms*, 205-212, 2002.

[14] S. Maruyama, H. Miyagawa, and H. Sakamoto. Improving Time and Space Complexity for Compressed Pattern Matching. *Proc. 17th Int. Sympo. Algorithms and Computation*, LNCS4288:484-493, 2006.

[15] C. Nevill-Manning, and I. Witten. Compression and Explanation Using Hierarchical Grammars. *Computer Journal*, 40(2/3):103–116, 1997.

[16] C. Nevill-Manning, and I. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artificial Intelligence Research*, 7:67–82, 1997.

[17] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211-222, 2003.

[18] H. Sakamoto, T. Kida, and S. Shimozono. A Space-Saving Linear-Time Algorithm for Grammar-Based Compression. *Proc. 11th International Symposium on String Processing and Information Retrieval*, LNCS3109:218-229, 2004.

[19] H. Sakamoto. A Fully Linear-Time Approximation Algorithm for Grammar-Based Compression. *J. Discrete Algorithms*, 3(2-4):416-430, 2005.

[20] J.A. Storer, and T.G. Szymanski. The Macro Model for Data Compression. *Proc. 10th Ann. Sympo. Theory of Computing*, 30-39, 1978.

[21] T.A. Welch. A Technique for High Performance Data Compression. *IEEE Comput.*, 17:8-19, 1984.

[22] E.-H. Yang, and J.C. Kieffer. Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform–Part One: without Context Models. *IEEE Trans. Inform. Theory*, 46(3):755-777, 2000.

[23] J. Ziv, and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inform. Theory*, IT-23(3):337-349, 1977.

[24] J. Ziv, and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inform. Theory*, 24(5):530-536, 1978.