Algorithms **2012**, *5*, 214-235; doi:10.3390/a5020214

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

An Online Algorithm for Lightweight Grammar-Based Compression

Shirou Maruyama [1], Hiroshi Sakamoto [2] and Masayuki Takeda [1,*]

[1] Department of Informatics, Kyushu University, 774 Motooka Nishi-ku Fukuoka-shi, Fukuoka, Japan; E-Mail: shiro.maruyama@i.kyushu-u.ac.jp

[2] Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu Iizuka-shi Fukuoka, Japan; E-Mail: hiroshi@ai.kyutech.ac.jp

[*] Author to whom correspondence should be addressed; E-Mail: takeda@inf.kyushu-u.ac.jp; Tel.: +81-92-802-3602; Fax: +81-92-802-3710.

Received: 30 January 2012; in revised form: 26 March 2012 / Accepted: 28 March 2012 / Published: 10 April 2012

Abstract: Grammar-based compression is a well-studied technique to construct a context-free grammar (CFG) deriving a given text uniquely. In this work, we propose an online algorithm for grammar-based compression. Our algorithm guarantees $O(\log^2 n)$-approximation ratio for the minimum grammar size, where n is an input size, and it runs in input linear time and output linear space. In addition, we propose a practical encoding, which transforms a restricted CFG into a more compact representation. Experimental results by comparison with standard compressors demonstrate that our algorithm is especially effective for highly repetitive text.

Keywords: lossless compression; grammar-based compression; online algorithm; approximation algorithm

1. Introduction

Grammar-based compression [1] finds a small *context-free grammar* (*CFG*) that generates a given string uniquely. Let us express grammar-based compression by an intuitive example. If a string w contains many occurrences of a substring γ, we can replace all of them by a single variable A which is associated with γ like $A \rightarrow \gamma$. The text is thus compressed to a shorter one according to the frequency of γ. The expression of grammar-compressed strings is simple, yet it is powerful

because they can derive a string of its exponential length. In fact, it was recently reported that grammar-based and LZ77-based [2] compressors can achieve effective compression for *highly-repetitive text* [3–5] by comparison with entropy-based encoders. Grammar-compressed strings are also suitable for accelerating string processing, for examples, combinatorial pattern matching [6–11], edit-distance computation [12], q-gram computation [13,14], mining characteristic patterns [15,16], and so on. We note above compressed string algorithms, mainly from theoretical approaches, are considered on the *straight-line programs* (*SLPs*). An SLP is a CFG in the Chomsky normal form that derives a single string. This situation is reasonable because any CFG deriving a single string can be straightforwardly converted into an SLP without paying significant time and space penalties. The smaller a given CFG is, the more these compressed string algorithms are accelerated to running time, so they desire effective compression algorithms which can guarantee to produce small CFGs in any case. Also, we should pay attention to working costs for the compression phase when we need to consider total time and space. One of our interests is, therefore, to translate an input string into a good CFG under the conditions of efficient working time and space.

In a theoretical sense, the NP-hardness and its approximation hardness for finding the smallest CFG from the input text was proved [17]. For this reason, there exist many characteristic compression algorithms proposed so far. In the grammar-based compression, some algorithms based on greedy strategies are known to achieve high compression ratios for the real-world texts, e.g., SEQUITUR [18], RE-PAIR [19], GREEDY [20], LFS2 [21], *etc*. Their upper bound of approximation ratios were theoretically analyzed in [22]. The best approximation ratio among greedy algorithms is $O((n/\log n)^{1/2})$, where n is the input string length (In this paper, $\log$ stands for $\log_2$). On the other hand, several algorithms achieving a logarithmic approximation ratio were proposed. For the minimum grammar size g_*, the first $O(\log(n/g_*))$-approximation algorithm was developed by Charikar *et al* [22]. Independently, Rytter [23] presented another $O(\log(n/g_*))$-approximation algorithm using the suffix tree. Sakamoto [24] also proposed a linear-time $O(\log(n/g_*))$-approximation algorithm based on RE-PAIR. However, these algorithms require $\Omega(n)$ space and this weakness prevents us from applying them to huge texts. This space complexity was improved by several multi-pass algorithms over read/write streams. Sakamoto *et al*. [25] proposed the LCA algorithm that requires $O(g_* \log g_*)$ space with linear running time and $O(\log n \log g_*)$-approximation ratio. LCA was modified to achieve $O(\log n \log^* n)$-approximation ratio within $O(n \log^* n)$ running time [26], where $\log^* n$, called the iterated logarithmic function, is the number of times the $\log$ function is applied to n to produce a constant. On the other hand, Gagie and Gawrychowski [27] proposed $O(\min(g_*, \sqrt{n \log n}))$-approximation algorithm in a streaming model where the algorithm works in constant space and logarithmic passes over a constant number of streams. Here, we must point out that these lightweight algorithms require large external memory space for managing read/write streams, and thus the practical running time is affected by the I/O response time. Moreover, the main results of approximation algorithms almost consist of theoretical achievements and their practical compressive performance is either not known or worse than popular compression programs.

Because of these factors, we assume more empirical situation. Many practical data compressors mandate linear running time in the length of the input string. Ideally, a compressor should also be online; that is, it processes the characters of the input string from left to right, one by one, with no need to know

the whole string beforehand. Preferably, the space consumption throughout compression processing should depend on the size of the compressed string, not the size of the string being compressed. We thus focus on the compression with a restricted resource and develop an online algorithm preserving a good approximation ratio. The proposed algorithm is based on LCA. Thanks to its simplicity, LCA does not require special data structures and it runs in linear time and an economical space. The required space by proposed algorithm is $O(g_* \log^2 n)$ and the approximation ratio is $O(\log^2 n)$. The main task of LCA is to replace long and frequent substrings with a common nonterminal within a smaller work space than $\Omega(n)$. Thus, an obtained CFG is much smaller with highly repetitive texts, so we implement the online LCA algorithm as a more practical compressor. To do this, we introduce a practical encoding technique that cuts off the constant factor of the output grammar size. The proposed encoding is based on the binary tree representation of CFGs. The space complexity of the improved LCA algorithm is proportional to the size of the produced CFG. Therefore, it can be expected that the smaller work space is required when the given text is extremely compressible. Our experiments show that the online LCA achieves effective compression for highly-repetitive text compared with other standard compressors, and the space consumption is smaller than the input size.

2. Preliminary

This section gives the notations and definitions for string and grammar-based compression.

2.1. Basic Notations

We assume a finite *alphabet* Σ for the symbols forming input strings throughout this paper. 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 $w \in \Sigma^*$ is denoted by $|w|$, and the cardinality of a set C is similarly denoted by $|C|$.

Strings x and z are said to be a *prefix* and *suffix* of the string $w = xyz$, respectively. Also, x, y, z are called *substrings* of w. The ith symbol of w is denoted by $w[i]$. For integers i, j with $1 \leq i \leq j \leq |w|$, the substring of w from $w[i]$ to $w[j]$ is denoted by $w[i, j]$.

A *repetition* is a string x^k for a symbol x and an integer $k \geq 2$. A repetition $w[i, j] = x^k$ is *maximal* if $w[i-1], w[i+1] \neq x$. It is simply referred to by x^+ if the length is unnecessary. Substrings $w[i, j]$ and $w[k, \ell]$ are *overlapping* if $i < k \leq j < \ell$. A string of length two is called a *pair*.

2.2. Grammar-Based Compression

A *context-free grammar* (*CFG*) is a quadruple $G = (\Sigma, N, \mathcal{D}, S)$ of disjoint finite alphabets Σ and N, a finite set (a dictionary) $\mathcal{D} \subset N \times (N \cup \Sigma)^*$ of *production rules*, and the *start symbol* $S \in N$. Symbols in N are called *nonterminals*. A production rule $A \to b_1, \ldots, b_k$ in $\mathcal{D}$ *derives* $\beta \in (\Sigma \cup N)^*$ *from* $\alpha \in (\Sigma \cup N)^*$ by replacing an occurrence of $A \in N$ in α with $b_1, \ldots, b_k$, denoted by $\alpha \Rightarrow \beta$. Similarly we say that $\mathcal{D}$ derives β from α provided $\alpha \Rightarrow^* \beta$, where $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$. If a string is derived from the start symbol, we also say that the CFG derives the string. In this paper, we assume that any CFG is *admissible* [1]; that is G derives just one string in Σ^* and for each nonterminal $A \in N$, exactly one production rule $A \to \alpha$ is defined in $\mathcal{D}$. We also assume that any $A \in N$ is *appropriate*, that is, $A \to \alpha, B \to \alpha \in \mathcal{D}$ implies $A = B$. The *size* of G is the total length of

strings on 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
Input: A string $w \in \Sigma^*$.
Output: An admissible CFG G *that derives* w.
Measure: The size of G.

In the following, we assume that every admissible CFG is restricted such that the length of right hand side of any production rule is two. Note that for any CFG G, there is an equivalent restricted CFG G' whose size is at most $2|G|$. Thus this restriction is reasonable.

An important relation is known to exist between an admissible CFG and the following factorization. The *LZ-factorization* $LZ(w)$ of w into $f_1, \ldots, f_k$, where $f_1 = w[1]$, and for each $1 < \ell \leq k$, f_ℓ is the longest prefix of $suf_\ell = f_\ell, \ldots, f_k$ that appears in $f_1, \ldots, f_{\ell-1}$, and otherwise $f_\ell = suf_\ell[1]$. 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.

Theorem 1 (Rytter [23]). *For any string* w *and its admissible CFG* G*, the inequality* $|LZ(w)| \leq |G|$ *holds.*

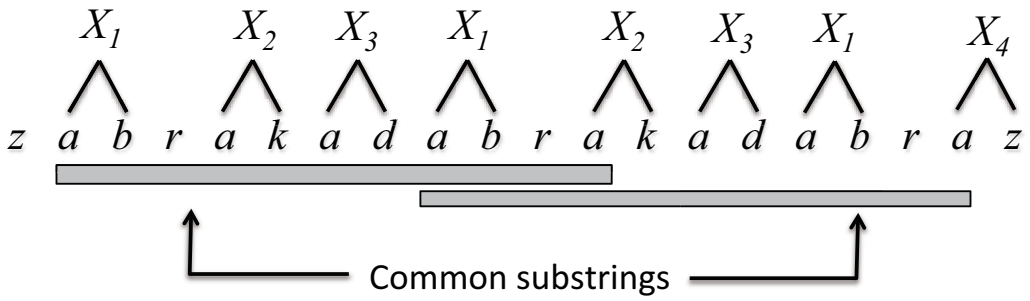
3. Compression Algorithm

This section presents our proposed algorithm and analyzes its performance.

3.1. Basic Idea

The basic task of the algorithm is to replace a pair XY occurring in a string by a new symbol Z and generate a production $Z \rightarrow XY$ to $\mathcal{D}$, where all occurrences of XY that are determined to be replaced are replaced by a same Z. We note, however, that not all occurrences of XY are replaced by Z. The critical task is to determine which occurrence of XY is replaced such that replaced pairs in common substrings are almost synchronized as shown in Figure 1. That is, the aim of this algorithm becomes to minimize the number of different nonterminals generated.

Figure 1. An example of replacing pairs. Our aim is to replace pairs that are almost synchronized in common substrings.



Here we explain the three decision rules for the replacement. The rules introduced in this paper are modified version of the Sakamoto *et al.*'s algorithm [25] to extend to our online compression in the next subsection.

The first rule (repetitive pair): Let a current string $\mathcal{S}$ contain a maximal repetition $\mathcal{S}[i,j] = a^k$. We generate $A \to aa \in \mathcal{D}$ for an appropriate nonterminal A, and replace $\mathcal{S}[i,j] = A^{k/2}$ if k is even, or replace $\mathcal{S}[i,j-1] = A^{(k-1)/2}$ if k is odd.

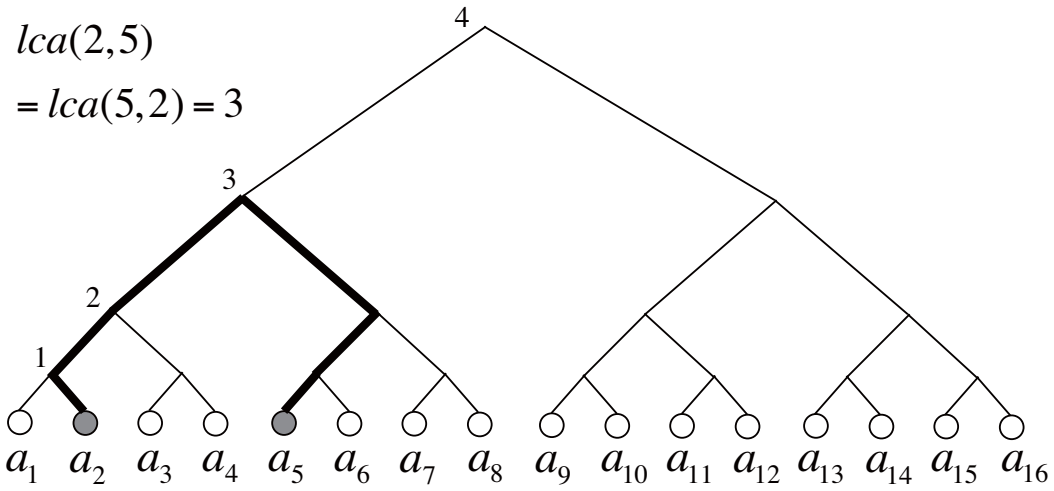
The second rule (minimal pair): We assume a total order over $\Sigma \cup N$; that is, any symbol is represented by an integer. If a current string contains a substring $A_iA_jA_k$ such that $j < i,k$, then the occurrence of A_j is called *minimal*. The second decision rule is to replace all such pairs A_jA_k in $A_iA_jA_k$ by an appropriate nonterminal.

In order to introduce the third decision rule, we explain the notion of the lowest common ancestor on a tree.

Definition 1. *Let p be a positive integer and $k = \lceil \log p \rceil$. The index tree T_p is the rooted, ordered complete binary tree whose leaves are labeled with $1, \ldots, 2^k$ from the left. The height of a node v refers to the number of edges in the longest path from v to a descendant of v. Then, the height of the lowest common ancestor of leaves i, j is denoted by lca(i,j) (We can get the lca for any leaves i and j of (virtual) complete binary tree in $O(1)$ time/space under RAM model [28].) for short.*

Figure 2 shows an example of the index tree and lowest common ancestor.

Figure 2. The (virtual) index tree T_{16} for $\Sigma \cup N = \{a_1, a_2, \ldots, a_{16}\}$.



The third rule (maximal pair): For a fixed order of alphabet, let a current string contain a substring $A_iA_jA_kA_\ell$ such that the integers i,j,k,ℓ are either increasing or decreasing in this order. If $lca(j,k) > lca(i,j), lca(k,\ell)$, then the occurrence of the middle pair A_jA_k is called *maximal*. The third decision rule is to replace all such pairs by an appropriate nonterminal.

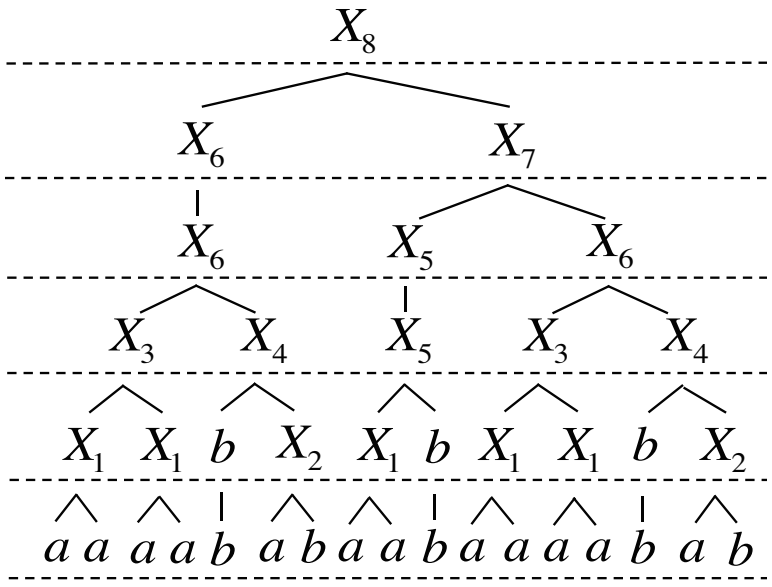
We call pairs replaced by the above rules *special pairs* which appear in almost synchronized position in the common substrings. Be careful that we need to set the priority of the decision rules because such cases possibly overlap and we cannot apply the repetitive and minimal rules simultaneously. For example, the substring $a_2a_1a_3a_3a_3$ contains such overlapping pairs. We therefore apply the repetitive and minimal rules in this order to keep uniqueness of the replacement. Indeed, no cases overlap with this priority.

If pairs $w[i-2,i-1]$ and $w[j+1,j+2]$ $(i<j)$ are special pairs and a substring $w[i,j]$ contains no special pairs, then we suitably determine replaced pairs in $w[i,j]$ with left priority, that is, if $j-i$ is odd

then the pairs $w[i, i+1], w[i+2, i+3], \ldots, w[j-1, j]$ are replaced, otherwise $w[i, i+1], w[i+2, i+3], \ldots, w[j-2, j-1]$. By this replacement, the length of the interval not replaced for a string becomes at most one.

In the case of offline compression, it is easy to generate a grammar by the replacement rules. After we replace pairs in a input string w, we recursively continue this process for a new string produced by replacing pairs until the replaced string becomes only one symbol as shown in Figure 3. We note the height of a parse tree is bounded by $O(\log n)$, where $n = |w|$, because the algorithm replace at least one of $w[i, i+1]$ or $w[i+1, i+2]$. In the next subsection, we will apply the basic idea to the online algorithm.

Figure 3. An example of a parse tree produced with our replacement rules.



3.2. Algorithmic Detail

The offline algorithm makes a bottom-up parse tree represented as a CFG. On the other hand, the online algorithm approximates the compression by simulating the left to right construction of a parse tree. To do this, we must determine a replaced pair by only a short substring. By the priority of the rules, we can determine that the pair for any position is the special pair or not by checking the rules simultaneously. In addition, the following lemma enable us to determine a replaced pair from the only a substring of length five.

Lemma 1. *Assume that replaced pairs in $w[1, i-1]$ are already determined, whether $w[i, i+1]$ becomes a replaced pair or not depends on the interval $w[i-1, i+3]$.*

Proof. We consider to decide whether the substring $w[i, i+2]$ includes the special pair under the assumption that $w[i-2, i-1]$ is already replaced. We first need to check a repetitive pair is included or not because of the strongest rule. For the case that $w[i, i+2] = a_r a_r a_s (r \neq s)$, we replace $w[i, i+1]$ as a repetitive pair. For the case that $w[i, i+2] = a_r a_s a_s (r \neq s)$, we preferentially select $w[i+1, i+2]$ as a replaced pair. We also pay attention to the case such that $w[i, i+3] = a_r a_s a_t a_t (s \neq t)$, then we forcibly replace $w[i, i+1]$ because $w[i+2, i+3]$ is the beginning of maximal repetition. For the minimal or

maximal pair, we can decide $w[i, i+1]$ is minimal and maximal pair by computing with $w[i-1, i+1]$ and $w[i-1, i+2]$, respectively. If $w[i+1, i+2]$ is minimal or maximal pair, then $w[i]$ is not selected as a replaced pair by the priority. Thus there is no conditional statement using the outside of the interval $w[i-1, i+3]$ for computing special pairs. □

From the Lemma 1, we describe the Algorithm 1 that is the statement to determine $w[i, i+1]$ is replaced or not. Note if $w[i, i+2]$ contains no special pair, we determine $w[i, i+1]$ as a replaced pair. By using Algorithm 1, it is easy to replace pairs in one pass over a string.

Algorithm 1 $replaced\_pair(w, i)$: a string w and a position i.

```
1: /* replaced_pair(w, i) decides a pair w[i, i+1] is replaced or not */
2: if w[i, i+1] is the repetitive pair then
3:     return true;
4: else if w[i+1, i+2] is the repetitive pair then
5:     /* w[i+1, i+2] is preferentially replaced. */
6:     return false;
7: else if w[i+2, i+3] is the repetitive pair then
8:     /* w[i, i+1] is forcibly replaced by the priority of the repetitive pair. */
9:     return true;
10: else if w[i, i+1] is the minimal or maximal pair then
11:     return true;
12: else if w[i+1, i+2] is the minimal or maximal pair then
13:     /* w[i+1, i+2] is preferentially replaced. */
14:     return false;
15: else
16:     /* w[i, i+2] contains no special pair. */
17:     return true;
18: end if
```

The compression algorithm determine replaced pairs in short buffers corresponding to each level of a parse tree. Let h is the height of the parse tree. We first prepare queues $q_1, q_2, \ldots, q_h$ implemented by circular buffers. Each q_i has a role as a buffer to store a segment of string w at the ith level of the parse tree, where the input string corresponds to first level of the tree. The number h of queues is bounded by $O(\log n)$ because of the height of parse tree. We define basic operations for such queues as follows:

- $enque(q_i, x)$: add the symbol x into the tail of the queue q_i.
- $deque(q_i)$: return the head of the queue q_i and remove it.
- $head(q_i)$: return the head of the queue q_i.
- $len(q_i)$: return the length of the queue q_i.

The head of the queue q_i is denoted by $q_i[0]$ and thus the tail corresponds to $q_i[len(q_i)-1]$. Each queue is used for deciding a replaced pair using the function $replaced\_pair(w, i)$ and its maximum length can be bounded by $O(1)$.

By the Lemma 1, we can restrict the maximum length of any queue by a constant longer than five.

For linear time compression, we must prepare another data structure $\mathcal{D}^R$ called *reverse dictionary*: $\mathcal{D}^R(x, y)$ returns a nonterminal z associated with the pair xy by $z \to xy \in \mathcal{D}$. In case $z \to xy \notin \mathcal{D}$, $\mathcal{D}^R(x, y)$ creates a new nonterminal symbol $z' \notin N$ and returns z'. For instance, if we have a dictionary $\mathcal{D} = \{X_1 \to a_1a_2, X_2 \to a_3a_1, X_3 \to a_2a_2\}$, $\mathcal{D}^R(a_3, a_1)$ returns X_2, $\mathcal{D}^R(a_1, a_1)$ creates a new nonterminal X_4 and returns it. If we use randomization, $\mathcal{D}^R(x, y)$ can be computed in $O(1)$ worst case time and inserting a new production rule can be achieved in $O(1)$ amortized time within $O(|\mathcal{D}|)$ space using dynamic perfect hashing [29].

Next we outline the online algorithm. We describe the online version of LCA in Algorithm 2 as well as its recursive function $insert\_symbol(q_i, x)$ in Algorithm 3. All queues are initialized to contain only *dummy symbol* $d \notin \Sigma \cup N$, which is required to compute the first pair at the each queue. In the lines 2–5 of Algorithm 2, input characters are enqueued to q_1 one by one. In Algorithm 3, if there is q_i such that $len(q_i) \geq 5$, the algorithm decides the replaced pair in $q_i[1, 3]$. In case $q_i[1, 2]$ is replaced by an appropriate nonterminal z, $q_i[0, 1]$ is dequeued and z is enqueued to q_{i+1}. In case $q_i[2, 3]$ is replaced by z, $q_i[0, 2]$ is dequeued and $q_i[1]z$ is enqueued to q_{i+1}. The symbol $q_i[2]$ in the first case and $q_i[3]$ in the second case are remaining in q_j to determine the next replaced pair after a new symbol is enqueued to q_i. Figure 4 describes the action of the function $insert\_symbol(q_i, x)$. The algorithm recursively continues the above process until all input characters are enqueued. As the post-processing, the symbols remaining in the queues $q_i, \ldots, q_h$ are replaced by appropriate nonterminals in the left to right order. Finally the produced dictionary is returned.

Algorithm 2 LCA-online.

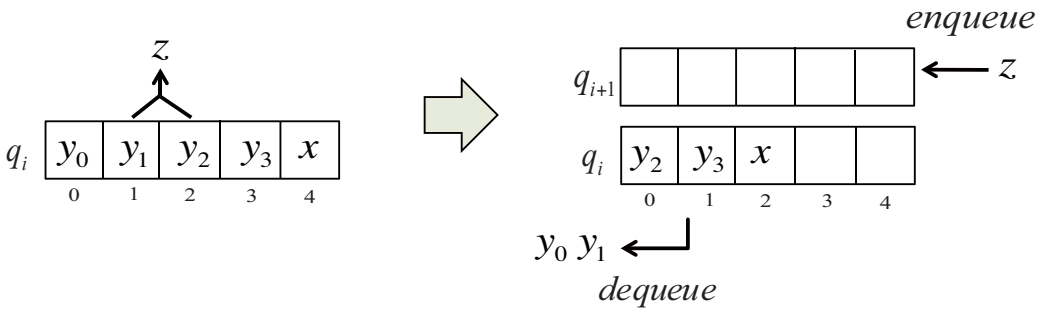
1: $\mathcal{D} := \emptyset$; initialize queues;
2: **repeat**
3: **input** a new character c;
4: $insert\_symbol(q_1, c)$;
5: **until** c is not the end of the inputs.
6: $i := 1$;
7: **while** q_i is not empty **do**
8: replace the symbols remained in $q_i[1, 4]$,
9: and then enqueue the string replaced in $q_i[1, 4]$ into q_{i+1};
10: $i := i + 1$;
11: **end while**
12: **output** $\mathcal{D}$;

Algorithm 3 $insert\_symbol(q_i, x)$: a queue q_i and a symbol x.

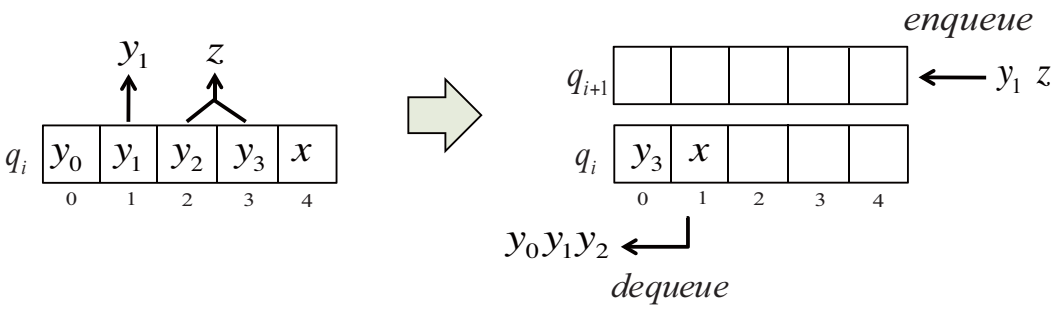
```
1:  enque(q_i, x);
2:  if len(q_i) ≥ 5 then
3:    if replaced_pair(q_i, 1) = true then
4:      deque(q_i);
5:      y_1 := deque(q_i); y_2 := head(q_i);
6:      z := D^R(y_1, y_2);
7:      D := {z → y_1 y_2} ∪ D;  /* update D */
8:      insert_symbol(q_{i+1}, z);
9:    else
10:     deque(q_i);
11:     y_1 := deque(q_i);
12:     insert_symbol(q_{i+1}, y_1);
13:     y_2 := deque(q_i); y_3 := head(q_i);
14:     z := D^R(y_2, y_3);
15:     D := {z → y_2 y_3} ∪ D;  /* update D */
16:     insert_symbol(q_{i+1}, z);
17:   end if
18: end if
```

Figure 4. The action of $insert\_symbol(q_i, x)$.

case 1 : $q_i[1,\ 2]$ is a replaced pair.



case 2 : $q_i[2,\ 3]$ is a replaced pair.



3.3. Performance Analysis

First, we estimate the running time of LCA-online. We use the following notation indicating the string enqueued to a queue.

Definition 2. *For each queue q_i, let $\mathcal{S}_i$ denote the string obtained by concatenating all symbols enqueued to q_i from left to right order.*

Note that $\mathcal{S}_i$ corresponds to the ith level of string in a parse tree produced by replacing pairs as shown in Figure 3. We first prove the following characteristic.

Theorem 2. *The running time of LCA-online is bounded by $O(n)$, where n is the input length.*

Proof. For any $\mathcal{S}_k$, the inequality $\frac{1}{2}|\mathcal{S}_k| \leq |\mathcal{S}_{k+1}| \leq \frac{2}{3}|\mathcal{S}_k|$ holds because the algorithm replaces at least one of $\mathcal{S}_k[i, i+1]$ and $\mathcal{S}_k[i+1, i+2]$. Therefore, $k \leq O(\log n)$ and the total number of symbols inserted into all queues is bounded by $O(n)$. In any queue, computing a replaced pair is $O(1)$ time because we can verify in $O(1)$ time whether or not $\mathcal{S}_k$ contains one of the repetitive, minimal, and maximal pairs. Also, computing the appropriate nonterminal for any pair can be computed in $O(1)$ time. Hence, the running time is bounded by $O(n)$ time. □

Next, we prove that the approximation ratio of LCA-online is *reasonable*. The approximation ratio of the compression algorithm is the upper bound of $\frac{g}{g_*}$ for the output grammar size g and the minimum grammar size g_* when arbitrary input string is given.

Definition 3. *Let $\mathcal{S}$ be a string and $\mathcal{S}[i, j] = \alpha$ be an occurrence of a substring α in $\mathcal{S}$. We call $\mathcal{S}[i, j]$ a* boundary occurrence *if $\mathcal{S}[i] \neq \mathcal{S}[i+1]$ and $\mathcal{S}[j] \neq \mathcal{S}[j-1]$.*

Definition 4. *Let S_t be a string enqueued to q_t. Then $R_t(i, j)$ is the shortest substring of S_{t+1} which derives a string containing $S_t[i, j]$.*

Lemma 2. *Let $\mathcal{S}_t[i_1, j_1] = \mathcal{S}_t[i_2, j_2] = \alpha$ be any boundary occurrences. For input string length n, there exists an integer $k \leq \log n$ such that $R_t(i_1 + k, j_1 - k) = R_t(i_2 + k, j_2 - k)$.*

Proof. Let us consider the index tree T_n. If a string $\alpha = a_{\ell_1}, a_{\ell_2}, \ldots, a_{\ell_m}$ of length m is a monotonic; *i.e.*, either $\ell_1 > \ell_2 > \ldots > \ell_m$ or $\ell_1 < \ell_2 < \ldots < \ell_m$, and $lca(\ell_1, \ell_2), lca(\ell_2, \ell_3), \ldots, lca(\ell_{m-1}, \ell_m)$ are monotonic, then m is bounded by $\log n$. Therefore, at least one of minimal pair or maximal pair must appear within $\log n$ consecutive symbols having no repetition. Thus, a prefix of $\mathcal{S}_t[i_1, j_1]$ longer than $\log n$ contains at least one of minimal/maximal pair; it also appears in $\mathcal{S}_t[i_2, j_2]$ at the corresponding position. Hence, the replacements of inside $\mathcal{S}_t[i_1, j_1]$ and $\mathcal{S}_t[i_2, j_2]$, *i.e.*, $\mathcal{S}_t[i_1 + k, j_1 - k]$ and $\mathcal{S}_t[i_2 + k, j_2 - k]$ completely synchronize. Hence, $R_t(i_1 + k, j_1 - k) = R_t(i_2 + k, j_2 - k)$ for $k \leq \log n$. □

Theorem 3. *The approximation ratio g/g_* of LCA-online is $O(\log^2 n)$, where g is the output grammar size, g_* is the minimum grammar size, and n is the length of the input string.*

Proof. We estimate the number of different nonterminals produced by LCA-online. Let $w_1, \ldots, w_m$ be the LZ-factorization of an input string w. Let $\#(w)$ denote the maximum number of different nonterminals generated in a single queue after the compression of w is completed. From the definition of

LZ-factorization, any factor w_i occurs in the prefix $w_1, \ldots, w_{i-1}$, or $|w_i| = 1$. First, we consider the case that w_i is a boundary occurrence. By Lemma 2, any two occurrences of w_i are respectively transformed to $\alpha\beta\gamma$ and $\alpha'\beta\gamma'$ such that $|\alpha| = |\alpha'|$, $|\gamma| = |\gamma'|$, and $|\alpha\gamma\alpha'\gamma'| = O(\log n)$. In the case that w_i is not a boundary occurrence, $w_i = a^+\lambda b^+$ for some string λ and repetitions a^+, b^+. For repetitions a^+ and b^+, the number of different nonterminals produced by the replacement of a^+ and b^+ is bounded by $O(1)$. If λ is a boundary occurrence, this case is the same as the one in which w_i is a boundary occurrence. If λ is not a boundary occurrence, $\lambda = c^+\lambda' d^+$ for some string λ', and $c \neq a$ and $d \neq b$. In this case, any occurrence of λ inside $a^+\lambda b^+$ is transformed to exactly same string. Thus, for a single queue, we can estimate $\#(w) = \#(w_1, \ldots, w_{m-1}) + O(\log n) = O(m \log n) = O(g_* \log n)$. Because the number of queues is at most $O(\log n)$, the size of the final dictionary is $O(g_* \log^2 n)$. □

Finally, we estimate the space complexity of our algorithm.

Theorem 4. *The space required by LCA-online is bounded by $O(g_* \log^2 n)$, where g_* is the minimum grammar size and n is the input string length.*

Proof. The number of queues is bounded by $O(\log n)$ and the length of any queue is $O(1)$. Thus, required space for the queues is $O(\log n)$. For the reverse dictionary, the space is bounded by the generated grammar size. By the Theorem 3, the space of the reverse dictionary is bounded by $O(g_* \log^2 n)$. Thus, the total space is bounded by $O(g_* \log^2 n)$. □

4. Encoding Technique

This section proposes a compact representation for a restricted CFG $G = (\Sigma, N, \mathcal{D}, S)$. In the following, we assume $\Sigma = \{1, 2, \ldots, \sigma\}$ for simplicity.

4.1. Encoded Representation of CFG

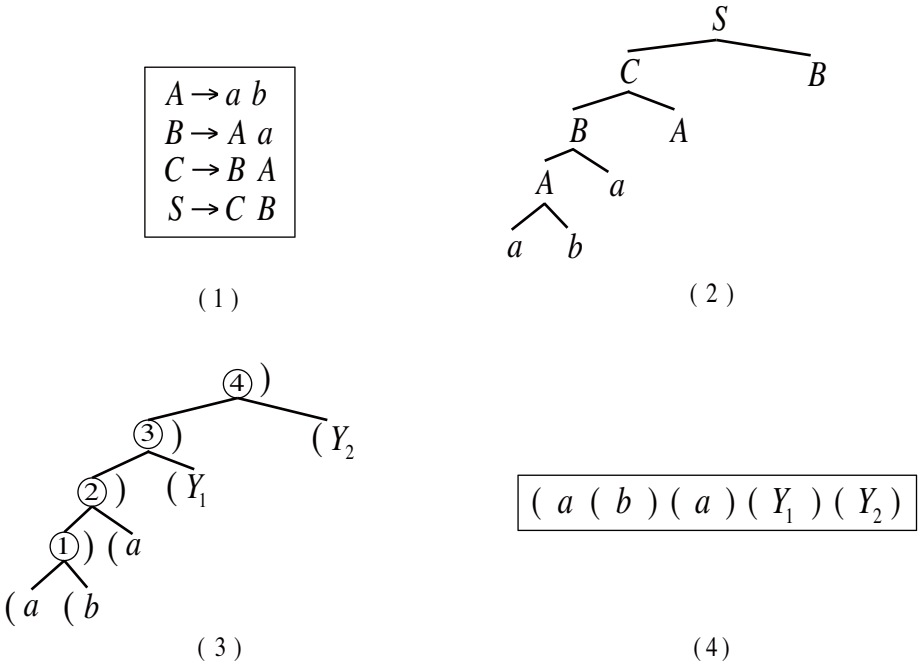
For the grammar G deriving w, we create the *partial parse tree* (This concept was introduced by Rytter [23].) $PTree(G)$, which is obtained by the following operation: Let T be the parsing tree for w by G. If T contains a maximal subtree rooted by $A \in N$ appearing in T at least twice, replace all occurrences of the subtree by a single node labeled by A except the leftmost occurrence of the subtree. When we continue this replacement, the final tree is denoted by $PTree(G)$. Figure 5 shows an example of the partial parse tree. $PTree(G)$ has g internal nodes and $g + 1$ leaves because $PTree(G)$ is a binary tree, where $g = |N|$. The construction of $PTree(G)$ can be in $O(g)$ time/space by expanding each nonterminal only once.

The skeleton of $PTree(G)$ is represented by a sequence of parentheses.

Let $x_1, x_2, \ldots, x_{2g+1}$ be a sequence of nodes sorted by post-order. We represent the sequence of nodes by $2g + 1$ parentheses as follows:

$$F[i] = \begin{cases} '(' & \text{if } x_i \text{ is a leaf} \\ ')' & \text{otherwise} \end{cases} \quad (1)$$

Figure 5. The example of the encoding process for CFG $G = (\{a, b\}, \{A, B, C, S\}, \mathcal{D}, S)$: (**1**) the dictionary $\mathcal{D}$; (**2**) the partial parse tree $PTree(G)$; (**3**) the parentheses representation of the tree and renamed labels where $Y_i = i + \sigma$; and (**4**) the encoded representation of G.



We then make a sequence of leaf labels of $PTree(G)$ to keep the information of the original string w. Let $E[1, g] \in N^g$ be the sequence of internal node labels of $PTree(G)$ in post-order. Let $M[1, g + 1] \in (\Sigma \cup N)^{g+1}$ be the sequence of leaf labels of $PTree(G)$ in post-order. We note that $E[1, g]$ is a permutation on N because every internal node has a different label from each other. Let $E^{-1}(N)$ be a function that maps any nonterminal $z \in N$ to the position i such that $E[i] = z$. Hereby, we define the sequence $L[1, g + 1]$, which consists of renamed nonterminals for M by the following:

$$L[i] = \begin{cases} E^{-1}(M[i]) + \sigma & (M[i] \in N) \\ M[i] & (M[i] \in \Sigma) \end{cases} \tag{2}$$

We then create the pair (F, L) as an encoded representation of CFG. Clearly the time/space to compute (F, L) is $O(g)$.

We estimate the bits of space required for (F, L). The space required for F is $2g + 1$ bits because F is the sequence over a binary alphabet representing g internal nodes and $g + 1$ leaves. Because L is the sequence over $\{1, 2, \ldots, g + \sigma\}$ whose length is $g + 1$, L can be represented in $(g + 1)\lceil \log(g + \sigma) \rceil$ bits of space. Thus, the total space for (F, L) is approximately $g\lceil \log(g + \sigma) \rceil + 2g$ bits. A naïve encoding represented by a sequence of right-hand sides of g production rules requires $2g\lceil \log(g+\sigma) \rceil$ bits of space. Thus, our representation reduces the space to almost half.

We note two array F and L can be combined into one array such that each symbol $L[i]$ is embedded after ith open parenthesis of F. The representation of the combined array has an advantage that the decoding processing can be done in one pass over the compressed text. We can also apply simple variable-length coding like LZW [30] for each element of L because the number of allocatable

nonterminals for any leaf node is limited by the number of internal nodes that appear before the leaf node in post-order. The efficiency of compression is further improved using such variable-length coding.

4.2. Decoding Process

We can decode the encoded representation of CFG because any nonterminal z in L indicates the position of the internal node corresponding to z. We describe the process in Algorithm 4. Scanning the compressed text (combined F and L) from the left to right, we can simulate the post-order traversal of the partial parse tree and restore the dictionary $\mathcal{D}$. To do this, we use a stack stk with two basic operations as follows:

- $push(stk, x)$: add symbol x into the top of the stack stk.
- $pop(stk)$: return the top of the stack stk and remove it.

Algorithm 4 Decode.

1: **input** a grammar size g, an alphabet size σ;
2: create an empty stack stk;
3: $i := 1$; $j := 1$; $k := 1$;
4: **while** $i \leq 2g + 1$ **do**
5: **input** a parenthesis $F[i]$;
6: **if** $F[i] =$'(' **then**
7: **input** a symbol $L[j]$;
8: **output** a string derived from $L[j]$ using $\mathcal{D}$;
9: $push(stk, L[j])$; $j := j + 1$;
10: **else**
11: $y_2 := pop(stk)$; $y_1 := pop(stk)$;
12: $z := k + \sigma$;
13: $\mathcal{D} := \{z \rightarrow y_1 y_2\} \cup \mathcal{D}$; /* update $\mathcal{D}$ */
14: $push(stk, z)$; $k := k + 1$;
15: **end if**
16: $i := i + 1$;
17: **end while**

When we decode $L[j]$ in the line 8, the required production rules are certainly contained in the current dictionary D by the characteristics of the partial parse tree. Thus, the algorithm can correctly output the original string by decoding the sequence $L[1, g + 1]$. The decoding time is bounded by $O(n)$ to output the original string, and the space is $O(g)$ to store the dictionary $\mathcal{D}$.

5. Experimental Results

We implemented three compressors based on the LCA algorithm, which are available from google code project (http://code.google.com/p/lcacomp/). The first, denoted by **LCA-online**, is the online algorithm of LCA proposed in Section 3. The second, denoted by **LCA-offline**, is a faithful

implementation of the offline LCA algorithm, which requires $o(g)$ space by using $O(n)$ external memory space, where g is the output grammar size and n is the input text size. Therefore, the compression speed of **LCA-offline** is affected by the I/O time. The third, denoted by **LCA-fast**, is another implementation of offline LCA, which requires $O(n)$ memory space and can thus achieve faster compression than **LCA-offline**.

For each generated CFG, two encoding methods are applied: one is the naïve encoding for the production rules and the other is the improved encoding presented in Section 4. Recall that the improved method requires $O(g)$ space. We distinguish between the naïve encoding and the improved one by the signs **:N** and **:I**, respectively. For example, **LCA-online:I** means the implementation of **LCA-online** with the improved encoding.

We compare our algorithms with other practical compressors. **LZW** [30] is a variant of LZ78-encoding [31], which we implemented. Our LZW implementation does not reset the codeword dictionary, unlike *compress* in UNIX programs. **gzip** (http://www.gzip.org) is based on LZ77-encoding [2] with limited window size. **bzip2** (http://www.bzip.org) is based on the block-sorting compression using the Burrows Wheeler Transform [32]. For gzip and bzip2, although we specified **-9** option to obtain their best compressive performances, those programs run in limited memory space because they output compressed texts before they have seen all of the input. **Re-Pair** [19] (http://www.cbrc.jp/∼rwan/en/restore.html) is an offline grammar-based compressor that recursively substitutes a new symbol for the most frequent pair. **LZMA** (p7zip) (http://p7zip.sourceforge.net/) is a powerful compressor based on the LZ77-encoding with unlimited window size. We set its window length as the input text length to achieve the best compressive performance. Table 1 summarizes the comparison in space usage and online/offline separation, where g is the output grammar size produced by LCAs, z is the number of output phrase parsed with LZW, and n is the input text length.

Table 1. Summary of comparison methods.

Method	Space usage	Online/Offline
LCA-online:I	$O(g)$	online
LCA-online:N	$O(g)$	online
LCA-offline:I	$O(g)$ + external space	offline
LCA-offline:N	$o(g)$ + external space	offline
LCA-fast:I	$O(n)$	offline
LCA-fast:N	$O(n)$	offline
LZW	$O(z)$	online
gzip -9	limited space	online
bzip2 -9	limited space	offline per blocks
Re-Pair	$O(n)$	offline
LZMA	$O(n)$	online

We used highly repetitive texts from repetitive corpus (Real) (http://pizzachili.dcc.uchile.cl/repcorpus.html), which consists of DNA sequences (*Escherichia_Coli, Para, Cere, influenza*), source codes (*coreutils, kernel*), and natural language texts (*einstein.de.txt, einstein.en.txt, world_leaders*). More detailed documentation is available from the Pizza & Chili (http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf). We also used general real world texts (*ENGLISH, XML*) from the Pizza & Chili corpus (http://pizzachili.dcc.uchile.cl/texts.html). ENGLISH is a natural language text collection written in English. XML is a structured text downloaded from http://dblp.uni-trier.de. Our environments are OS:CentOS 5.5 (64-bit), CPU:Intel Xeon E5504 2.0GHz (Quad)×2, Memory:144GB RAM. Our programs are written in the C language and compiled by gcc 4.1.2 optioned with -O3 optimization. We measure the processing time by the *time* command, and maximum memory usage in programs by the *memusage* command.

5.1. Comparison with Standard Compressors.

LCA-online:(I) is compared with other standard compressors in terms of the compression ratio, the consumption of memory space, and the compression time. Table 2(a) shows the result of compression ratio. Table 2(b) gives the result of main memory usage. Space usage is represented by the ratio to input text size. For general texts, the compression ratio of **LCA-online:(I)** is worse compared with other compressors. The repetitive substrings in typical texts are generally short, for example, single words in English texts, short tags in XML documents. Our algorithm seems to be weak to capture such short repetitive substrings. On the other hand, it achieves a higher compression ratio for the repetitive texts because our algorithm can replace long common substrings by the same nonterminals from the analysis of Section 3.3. **LZW** does not work well for the repetitive texts in spite of maintaining the dictionary. This is because LZW(LZ78) parsing does not guarantee to capture long and frequent substrings. **gzip** and **bzip2** also do not work well because they compress the input text in limited segments, not using the whole text. **Re-Pair** and **LZMA** use the whole text to powerful compression and thus they have a better compression ratio than ours. Therefore, as seen in Table 2(b), they require more memory space than the input text. On the other hand, the space requirement of ours and that of **LZW** depend on the output size. Especially, our compression ratios for repetitive texts are very high than that of LZW. Thus, the space usage becomes very small when the input text is sufficiently compressed.

Table 2(c) shows the average time per 1MiB for the compression processing. **LCA-online:(I)** can achieve fast compression independent of the kind of texts. But it is a little slower in the case of general texts than that of repetitive texts. This is because our implementation of the reverse dictionary seems to cause a little slowdown with increasing of the size of the dictionary. The other compressors, especially in **gzip**, **Re-Pair** and **LZMA**, are quite slow depending on the kind of text, especially in biological data with small alphabet.

Table 2. Experimental results for LCA-online *versus* standard compressors. (**a**) Compression ratio (percentage of the compressed size over the text size); (**b**) Main memory consumption (fraction of the text size); (**c**) Compression time (seconds per 1MiB).

Source		**online:I**	**LZW**	**gzip -9**	**bzip2 -9**	**Re-Pair**	**LZMA**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	12.43	25.46	37.64	26.98	9.60	4.43
Para	429,265,758	4.48	23.56	27.04	26.15	2.74	1.24
Cere	461,286,644	3.25	22.59	26.20	25.22	1.86	1.05
influenza	154,808,555	4.57	11.55	6.87	6.59	3.26	1.55
coreutils	205,281,778	5.23	22.24	24.32	16.02	2.54	1.99
kernel	257,961,616	2.18	24.13	26.90	21.74	1.10	0.82
einstein.de.txt	92,758,441	0.30	10.61	31.04	4.32	0.16	0.11
einstein.en.txt	467,626,544	0.17	6.68	35.00	5.17	0.10	0.07
world_leaders	46,968,181	3.40	12.48	17.65	6.94	1.79	1.39
General Text	Size (bytes)						
ENGLISH	209,715,200	40.86	33.06	37.64	28.07	31.79	21.12
XML	209,715,200	23.64	17.84	17.12	11.35	16.67	12.07

(**a**)

Source		**online:I**	**LZW**	**gzip -9**	**bzip2 -9**	**Re-Pair**	**LZMA**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	0.81	1.59	0.0065	0.062	26.98	10.24
Para	429,265,758	0.26	1.24	0.0017	0.016	24.43	10.02
Cere	461,286,644	0.21	1.46	0.0016	0.015	25.00	10.00
influenza	154,808,555	0.41	0.72	0.0047	0.045	25.06	10.16
coreutils	205,281,778	0.38	1.23	0.0036	0.034	25.03	10.22
kernel	257,961,616	0.23	1.48	0.0028	0.027	24.39	9.76
einstein.de.txt	92,758,441	0.15	0.61	0.0079	0.076	23.74	9.54
einstein.en.txt	467,626,544	0.033	0.42	0.0016	0.017	22.42	9.87
world_leaders	46,968,181	0.38	0.96	0.016	0.15	26.79	9.73
General Text	Size (bytes)						
ENGLISH	209,715,200	2.17	1.95	0.0035	0.034	27.00	10.00
XML	209,715,200	1.53	1.05	0.0035	0.034	25.00	10.00

(**b**)

Table 2. *Cont.*

Source		**online:I**	**LZW**	**gzip -9**	**bzip2 -9**	**Re-Pair**	**LZMA**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	0.11	0.14	1.56	0.16	1.65	1.76
Para	429,265,758	0.11	0.16	1.57	0.15	1.21	1.35
Cere	461,286,644	0.091	0.19	1.48	0.15	1.10	1.19
influenza	154,808,555	0.082	0.13	0.38	0.35	0.67	0.35
coreutils	205,281,778	0.14	0.18	0.12	0.20	0.94	0.37
kernel	257,961,616	0.14	0.20	0.11	0.15	0.80	0.72
einstein.de.txt	92,758,441	0.12	0.14	0.12	0.33	0.63	0.15
einstein.en.txt	467,626,544	0.11	0.16	0.11	0.34	0.67	0.16
world_leaders	46,968,181	0.076	0.10	0.089	0.11	0.50	0.21
General Text	Size (bytes)						
ENGLISH	209,715,200	0.21	0.22	0.18	0.16	3.42	0.92
XML	209,715,200	0.18	0.15	0.06	0.23	1.76	0.42

(**c**)

By these results, we can say **LCA-online:(I)** has practical properties for compressing huge highly repetitive texts with economical space, fast compression, and powerful compressive performance.

5.2. Comparison with Different Variations on LCA

Table 3(a), Table 3(b) and Table 3(c) show the compression ratio, the maximum memory usage, and the compression time within the variations of LCA, respectively. In Table 3(a) and Table 3(c), we can see that the improved encoding brings more efficient compression than the naïve encoding, and the processing time of the improved encoding is almost the same as that of the naïve encoding. On the other hand, regarding the grammar size, there is really not much difference between the online and offline algorithm. From Table 3(b) and Table 3(c), the running time of **LCA-online** is almost the same as that of **LCA-offline** and a bit slower than **LCA-fast**. However, we recall that the compression speed of **LCA-offline** depends on the I/O time of the computing environment; and **LCA-fast** always needs more memory space than the input text. In addition, **LCA-online** has an advantage to enable us incremental compression.

Table 3. Experimental results for LCA variations. (**a**) Compression ratio (percentage of the compressed size over the text size); (**b**) Main memory consumption (fraction of the text size); (**c**) Compression time (seconds per 1MiB).

Source		**online:I**	**online:N**	**offline:I**	**offline:N**	**fast:I**	**fast:N**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	12.43	24.58	12.21	24.19	12.21	24.19
Para	429,265,758	4.48	8.69	4.39	8.53	4.39	8.53
Cere	461,286,644	3.25	6.39	3.17	6.24	3.17	6.24
influenza	154,808,555	4.57	8.99	4.48	8.83	4.48	8.83
coreutils	205,281,778	5.23	10.05	5.22	10.05	5.22	10.05
kernel	257,961,616	2.18	4.17	2.17	4.16	2.17	4.16
einstein.de.txt	92,758,441	0.30	0.58	0.30	0.57	0.30	0.57
einstein.en.txt	467,626,544	0.17	0.33	0.17	0.33	0.17	0.33
world_leaders	46,968,181	3.40	6.70	3.41	6.71	3.41	6.71
General Text	Size (bytes)						
ENGLISH	209,715,200	40.86	79.37	40.41	78.538	40.41	78.54
XML	209,715,200	23.64	45.50	23.84	45.85	23.84	45.85

(**a**)

Source		**online:I**	**online:N**	**offline:I**	**offline:N**	**fast:I**	**fast:N**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	0.81	0.81	0.51	0.30	4.37	4.37
Para	429,265,758	0.26	0.26	0.17	0.078	4.15	4.15
Cere	461,286,644	0.21	0.21	0.13	0.036	4.09	4.09
influenza	154,808,555	0.41	0.41	0.20	0.095	4.09	4.09
coreutils	205,281,778	0.38	0.38	0.22	0.08	4.16	4.16
kernel	257,961,616	0.23	0.23	0.098	0.057	4.07	4.07
einstein.de.txt	92,758,441	0.15	0.15	0.15	0.15	4.07	4.07
einstein.en.txt	467,626,544	0.032	0.032	0.031	0.031	3.81	3.81
world_leaders	46,968,181	0.38	0.38	0.31	0.31	4.20	4.20
General Text	Size (bytes)						
ENGLISH	209,715,200	2.17	2.17	1.51	0.58	6.00	6.00
XML	209,715,200	1.53	1.53	0.92	0.26	5.54	5.54

(**b**)

Table 3. *Cont.*

Source		**online:I**	**online:N**	**offline:I**	**offline:N**	**fast:I**	**fast:N**
Repetitive Text	Size (bytes)						
Escherichia_Coli	112,689,515	0.11	0.12	0.11	0.11	0.086	0.083
Para	429,265,758	0.11	0.12	0.099	0.11	0.075	0.074
Cere	461,286,644	0.091	0.10	0.095	0.10	0.072	0.071
influenza	154,808,555	0.082	0.10	0.085	0.099	0.065	0.064
coreutils	205,281,778	0.14	0.16	0.13	0.13	0.10	0.10
kernel	257,961,616	0.14	0.16	0.12	0.12	0.098	0.098
einstein.de.txt	92,758,441	0.12	0.11	0.11	0.11	0.086	0.086
einstein.en.txt	467,626,544	0.11	0.12	0.11	0.11	0.087	0.087
world_leaders	46,968,181	0.076	0.084	0.078	0.079	0.060	0.059
General Text	Size (bytes)						
ENGLISH	209,715,200	0.21	0.16	0.18	0.17	0.15	0.14
XML	209,715,200	0.18	0.14	0.16	0.15	0.13	0.12

(**c**)

By these results, **LCA-online** has sufficient performance compared with the offline version, and the proposed encoding is very effective at CFG representation.

6. Summary

We developed an online algorithm for grammar-based compression. Our algorithm not only guarantees a reasonable approximation ratio for the minimum grammar, it also achieves effective compression for highly repetitive text, practically.

As future work, we will apply our grammar to string processing over compressed texts. For example, *compressed pattern matching* [33], *grammar-based self-index* [34,35], random accessible data structure [36] and so on. One property of our grammar is that the height of the parse tree is bounded by $O(\log n)$; another property is that our algorithm can find long common substrings without $\Omega(n)$ space data structures. These properties would be suitable for such compressed string processing.

Acknowledgements

The authors would like to thank the anonymous reviewers for their careful reading. This work was supported by JST PRESTO program and Grants-in-Aid for Young Scientists (A), MEXT (No. 23680016).

References

1. Kieffer, J.; Yang, E.H. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory* **2000**, *46*, 737–754.

2. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.
3. Sirẽn, J.; Välimäki, N.; Mäkinenn, V.; Navarro, G. Run-Length Compressed Indexes are Superior for Highly Repetitive Sequence Collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrievals (SPIRE '08)*, Melbourne, Australia, 10–12 November 2008; pp. 164–175.
4. Claude, F.; Feriña, A.; Martinez-Prieto, M.A.; Navarro, G. Compressed q-Gram Indexing for Highly Repetitive Biological Sequences. In *Proceedings of the 10th IEEE International Conference on Bioinformatics and Bioengineering (BIBE '10)*, Philadelphia, PA, USA, 31 May–3 June 2010; pp. 86–91.
5. Kreft, S.; Navarro, G. Self-Indexing Based on LZ77. In *Proceedings of the 22th Annual Symposium on Combinatorial Pattern Matching (CPM '11)*, Palermo, Italy, 27–29 June 2011; pp. 285–298.
6. Karpinski, M.; Rytter, W.; Shinohara, A. An efficient pattern matching algorithm strings with short descriptions. *Nordic J. Comput.* **1997**, *4*, 172–186.
7. Miyazaki, M.; Shinohara, A.; Takeda, M. An Improved Pattern Matching Algorithm for Strings in Terms of Straight-Line Programs. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97)*, CPM 97, Aarhus, Denmark, 30 June–2 July 1997; volume 1264, pp. 1–11.
8. Cégielski, P.; Guessarian, I.; Lifshits, Y.; Matiyasevich, Y. Window Subsequence Problems for Compressed Texts. In *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, St. Petersburg, Russia, 8–12 June 2006; pp. 127–136.
9. Lifshits, Y. Processing Compressed Texts: A Tractability Border. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM '07)*, London, Canada, 9–11 July 2007; volume 4580, pp. 228–240.
10. Tiskin, A. Towards Approximate Matching in Compressed Strings. In *Proceedings of the 6th International Computer Science Symposium in Russia (CSR '11)*, St. Petersburg, Russia, 14–18 June 2011; volume 6651, pp. 401–414.
11. Yamamoto, T.; Bannai, H.; Inenaga, S.; Takeda, M. Faster Subsequence and Don't-Care Pattern Matching on Compressed Texts. In *Proceedings of the 22th Annual Symposium on Combinatorial Pattern Matching (CPM '11)*, Palermo, Italy, 27–29 June 2011; volume 6661, pp. 309–322.
12. Hermelin, D.; Landau, G.M.; Landau, S.; Weimann, O. A Unified Algorithm for Accelerating Edit-Distance Computation via Text-Compression In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS '09)*, Freiburg, Germany, 26–28 February 2009; pp. 529–540.
13. Goto, K.; Bannai, H.; Inenaga, S.; Takeda, M. Fast q-Gram Mining on SLP Compressed Strings. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE '11)*, Pisa, Italy, 17–21 October 2011; volume 7028, pp. 278–289.
14. Goto, K.; Bannai, H.; Inenaga, S.; Takeda, M. Computing q-Gram Non-Overlapping Frequencies on SLP Compressed Texts. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '12)*, Spindlerv Mlýn, Czech Republic, 21–27 January 2012; pp. 301–312.

15. Inenaga, S.; Bannai, H. Finding Characteristic Substrings from Compressed Texts. In *Proceedings of the Prague Stringology Conference (PSC '09)*, Prague, Czech Republic, 31 August– 2 September 2009; pp. 40–54.
16. Matsubara, W.; Inenaga, S.; Ishino, A.; Shinohara, A.; Nakamura, T.; Hashimoto, K. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.* **2009**, *410*, 900–913.
17. Lehman, E.; Shelat, A. Approximation Algorithms for Grammar-Based Compression. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, San Francisco, CA, USA, 6–8 January 2002; pp. 205–212.
18. Nevill-Manning, C.; Witten, I. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res.* **1997**, *7*, 67–82.
19. Larsson, N.; Moffat, A. Off-line dictionary-based compression. *Proc. IEEE* **2000**, *88*, 1722–1732.
20. Apostolico, A.; Lonardi, S. Offline compression by greedy textual substitution. *Proc. IEEE* **2000**, *88*, 1733–1744.
21. Nakamura, R.; Inenaga, S.; Bannai, H.; Funamoto, T.; Takeda, M.; Shinohara, A. Linear-time off-line text compression by longest-first substitution. *Algorithms* **2009**, *2*, 1429–1448.
22. Charikar, M.; Lehman, E.; Liu, D.; Panigrahy, R.; Prabhakaran, M.; Sahai, A.; Shelat, A. The smallest grammar problem. *IEEE Trans. Inf. Theory* **2005**, *51*, 2554–2576.
23. Rytter, W. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.* **2003**, *302*, 211–222.
24. Sakamoto, H. A fully linear-time approximation algorithm for grammar-based compression. *J. Discret. Algorithms* **2005**, *3*, 416–430.
25. Sakamoto, H.; Kida, T.; Shimozono, S. A Space-Saving Linear-Time Algorithm for Grammar-Based Compression. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE '04)*, Padova, Italy, 5–8 October 2004; pp. 218–229.
26. Sakamoto, H.; Maruyama, S.; Kida, T.; Shimozono, S. A space-saving approximation algorithm for grammar-based compression. *IEICE Trans. Inf. Syst.* **2009**, *E92-D*, 158–165.
27. Gagie, T.; Gawrychowski, P. Grammar-Based Compression in a Streaming Model. In *Proceedings of the 4th International Conference on Language and Automata Theory and Applications (LATA '10)*, Trier, Germany, 24–28 May 2010; volume 6031, pp. 273–284.
28. Gusfield, D. *Algorithms on Strings, Trees, and Sequences*; Cambridge University Press: Cambridge, UK, 1997.
29. Dietzfelbinger, M.; Karlin, A.; Mehlhorn, K.; auf der Heide, F.M.; Rohnert, H.; Tarjan, R.E. Dynamic Perfect Hashing: Upper and Lower Bounds. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS '88)*, White Plains, New York, USA, 24–26 October 1988; pp. 524–531.
30. Welch, T. A technique for high-performance data compression. *IEEE Comput.* **1984**, pp. 8–19.
31. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
32. Burrows, M.; Wheeler, D. A Block Sorting Lossless Data Compression Algorithm; Technical Report 124, Digital Equipment Corporation, 1994.

33. Kida, T.; Matsumoto, T.; Shibata, Y.; Takeda, M.; Shinohara, A.; Arikawa, S. Collage systems: A unifying framework for compressed pattern matching. *Theor. Comput. Sci.* **2003**, *298*, 253–272.
34. Claude, F.; Navarro, G. Self-Indexed Grammar-Based Compression. *Fundam. Inform.* **2011**, *3*, 313–337.
35. Gagie, T.; Gawrychowski, P.; Puglisi, S.J. Faster Grammar-Based Self-Index. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA '12)*, A Coruña, Spain, 5–9 March 2012; pp. 273–284.
36. Bille, P.; Landau, G.M.; Raman, R.; Sadakane, K.; Satti, S.R.; Weimann, O. Random Access to Grammar-Compressed Strings. In *Proceedings of the 22th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '11)*, San Francisco, California, USA, 23–25 January 2011; pp. 373–389.

© 2012 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/.)