Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Discrete Algorithms 3 (2005) 416–430
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

A fully linear-time approximation algorithm for grammar-based compression

Hiroshi Sakamoto

Department of Artificial Intelligence, Kyushu Institute of Technology, Kawazu 680-4, Iizuka 820-8502, Japan

Available online 18 September 2004

Abstract

A linear-time approximation algorithm for the *grammar-based compression* is presented. This is an optimization problem to minimize the size of a context-free grammar deriving a given string. For each string of length n, the algorithm guarantees $O(\log \frac{n}{g_*})$ approximation ratio without suffix tree construction.

Keywords: Approximation algorithm; Linear-time algorithm; Lossless data compression; Grammar-based compression

1. Introduction

In this paper we design a simple approximation algorithm for the *grammar-based compression*. Given a string for an input, an output is a smallest context-free grammar that generates only the given string. Such a CFG is deterministic since every nonterminal is restricted to be derived from only one production. The complexity of such a combinatorial compression problem was firstly proved by Storer [17]. He considered the compression problem as a factorization of a given string by substrings and showed its NP-hardness. Moreover, De Agostino and Storer [2] introduced several online variations of this problem and showed that these problems are also NP-hard.

E-mail address: hiroshi@ai.kyutech.ac.jp (H. Sakamoto).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2004.08.016

The hardness of the approximability for the grammar-based compression was presented by Lehman and Shelat [12]. They showed that this problem is not approximable within a constant factor by an L-reduction from VERTEX COVER [1]. They also showed an interesting relation between the grammar-based compression and the so-called *semi-numerical problem* [9], which is an algebraic problem of minimizing the number of multiplications to compute given integers. Since no polynomial-time approximation ratio $o(\log n / \log\log n)$ is known for the semi-numerical problem and it is a special case of the grammar-based compression, improvement of the approximation ratio for the grammar-based compression beyond this threshold is at least as difficult.

On the other hand, the framework of the grammar-based compression can uniformly describe the dictionary-based coding schemes which are widely presented for real world text compression. For example, LZ78 [21] (including LZW [18]) and BISECTION [8] encodings are considered as algorithms in order to compute very restricted CFGs so that the length of the right hand of any production is at most 2. Other encodings for restricted CFG are also presented in [10,13,14,19]. For these practical algorithms, Lehman and Shelat [12] also showed their lower/upper bounds of the approximation ratio to the smallest CFGs. However, these bounds are relatively large to $O(\log n)$ ratio. For example, the lower and upper bound of LZW algorithm is $\Omega(n^{2/3}/\log n)$ and $O((n/\log n)^{2/3})$, respectively. The best proved approximation ratio was $O((n/\log n)^{1/2})$ for BISECTION algorithm. Other practical compression algorithms are presented in [6,7]

The first polynomial-time $O(\log n)$-approximation algorithms were produced by Charikar et al. [3] and Rytter [15], independently. Charikar et al. proposed an almost linear-time approximation algorithm using the notion of α-*balanced* strings. Their algorithm guarantees $O(\log \frac{n}{g_*})$ approximation ratio, where g_* is the size of the smallest grammar. Rytter's algorithm constructs a suffix tree for an input string and divides the string into the *LZ-factorization*. His algorithm is also the first linear-time approximation preserving the $O(\log \frac{n}{g_*})$ ratio.

In this paper we also propose a linear-time $O(\log \frac{n}{g_*})$-approximation algorithm using a simpler data structure for huge data compression. Our algorithm is based on the RE-PAIR encoding scheme by Larsson and Moffat [10]. As was shown in their paper, they experiment with the algorithm for large data and showed that RE-PAIR encoding is simple and space-efficient. The strategy of RE-PAIR is the recursive replacements of all pairs like ab in an input string according to their frequency. This encoding is also included in the framework of the grammar-based compression, while only the lower bound $O(\sqrt{\log n})$ of its approximation ratio is known [11]. Its nontrivial upper bound is still an important open problem.

To compute the frequency of all pairs, we assume a simple data structure. The assumed structure is a doubly-linked list for an input string so that any ith symbol is linked with $(i-1)$th and $(i+1)$th symbols, and any ith pair ab is linked with $(i-1)$th and $(i+1)$th occurrences of ab. This linked list can be easily constructed in linear time. Using this data structure, we can execute the compression and counting processes simultaneously. This is the reason that the running time of the algorithm is bounded by $O(n)$.

The main idea for our algorithm is the following. Assume that a string contains nonoverlapping intervals X and Y which represent a same substring. The aim of our algorithm is to compress them into some intervals which have a common substring as long as possible.

More precisely, the aim is that X and Y are compressed into $X' = \alpha\beta\gamma$ and $Y' = \alpha'\beta\gamma'$ such that the length of disagreement strings $\alpha\gamma$ and $\alpha'\gamma'$ are bounded by a constant. If this encoding is obtained for all same intervals, then the input is expected to be compressed in a sufficiently short string by successively applying this process to the resulting intervals X' and Y'.

In case X and Y are partitioned by some delimiter characters on their both sides, it is easy to compress them into a same string by RE-PAIR like algorithm. However X and possibly Y are generally overlapping with other intervals which represent other different substrings. The goal is to construct a linear-time algorithm for the required encoding without suffix tree construction. Let an input string contain an occurrence of a pair ab. If the ab is replaced by a nonterminal A, then our algorithm does not replace the A in the same loop. We then call our algorithm LEVELWISE-REPAIR by this strategy.

The approximation ratio is obtained from the comparison with the size of the output grammar G and the number of the LZ-factorization $LZ(w)$ for an input string w. By using the result $|LZ(w)| \leqslant |G_*|$ for the minimum grammar G_* [15], we prove the approximation ratio $O(\log \frac{n}{g_*})$ for our algorithm. This ratio is an improvement of the result in [16].

This paper is organized as follows. We define some notations for CFG and the grammar-based compression in Section 2. In Section 3, we propose the approximation algorithm as well as its procedures. We also give the data structures the *doubly-linked list* and *priority queue* to get constant-time access to any occurrence of pair. In Section 4, we show that the approximation ratio is $O(\log \frac{n}{g_*})$ and the running time is $O(n)$ for any input of length n and the size g_* of a smallest compression. The result is concluded in Section 5.

2. Preliminaries

We assume the following standard notations and definitions concerned with strings. An *alphabet* is a finite set of symbols. Let Σ be an alphabet. The set of all strings of length i over Σ is denoted by Σ^i and the length of a string w is denoted by $|w|$.

If $w = \alpha\beta\gamma$, then we call β a *substring* in w. An *interval* $w[i, j]$ is a substring β in $w = \alpha\beta\gamma$ associated with its occurrence position such that $i = |\alpha| + 1$ and $j = |\beta|$, where $1 \leqslant i \leqslant j \leqslant |w|$. Specially, in case $i = j$, the interval is denoted by $w[i]$. If $w[i, j]$ and $w[i', j']$ represent a same substring, it is denoted by $w[i, j] = w[i', j']$. An expression $\sharp(\alpha, \beta)$ denotes the number of occurrences of a string α in a string β. For example, $\sharp(ab, abababba) = 3$ and $\sharp(bab, abababba) = 2$.

An interval $w[i, j] = x^k$ for a symbol x is called a *repetition*. In particular, in case $w[i-1], w[j+1] \neq x$, we may write $w[i, j] = x^+$ if we do not need to specify the length k. Intervals $w[i, j]$ and $w[i', j']$ $(i < i')$ are called to be *overlapping* if $i' \leqslant j < j'$ and to be *independent* if $j < i'$.

An interval $w[i, i+1]$ $(1 \leqslant i \leqslant |w|-1)$ is called a *segment*. For any two symbols $a, b \in \Sigma$, if ab is a substring in a string w, then ab is said to be a *pair* in w. If $w[i, i+1] = ab$ for a pair ab and a segment $w[i, i+1]$, it is said that $w[i, i+1]$ is a segment of ab. For a segment $w[i, i+1]$, two segments $w[i-1, i]$ and $w[i+1, i+2]$ are called the *left* and *right* segments of $w[i, i+1]$, respectively.

A *context-free grammar* (CFG) is a 4-tuple $G = (\Sigma, N, P, S)$, where Σ and N are disjoint alphabets, P is a finite set of binary relations called *production rules* between N and the set of strings over $\Sigma \cup N$, and $S \in N$ is called the *start symbol*. Elements in N are called *nonterminals*. A production rule in P represents a replacement rule written by $A \to B_1 \cdots B_k$ for some $A \in N$ and $B_i \in \Sigma \cup N$.

We assume that any CFG considered in this paper is *deterministic*, that is, for each $A \in N$, exactly one production $A \to \alpha$ exists in P. Thus, the language $L(G)$ defined by G is a singleton set.

The *size* of G, denoted by $|G|$, is the total length of right sides of all production rules. The grammar-based compression problem is then defined as follows.

Problem 1 (*Grammar-Based Compression*).

INSTANCE: A string w.
SOLUTION: A deterministic CFG G for w.
MEASURE: The size $|G|$ of G.

For each CFG G, we can obtain a CFG G' in Chomsky normal form such that $L(G) = L(G')$ and $|N'| \leqslant 2|N|$. Thus we can replace the measure of the problem by the size of a set of nonterminals.

3. Approximation algorithm

We present the approximation algorithm LEVELWISE-REPAIR for the grammar-based compression in Fig 1. This algorithm calls two procedures *repetition*(,) in Fig. 2 and *arrangement*(,) in Fig. 3.

3.1. Outline of the algorithm

The algorithm contains two procedures *repetition* and *arrangement*. They are called by the algorithm for each execution of the outer-loop. The task of *repetition* is to replace

```
1 Algorithm LEVELWISE-REPAIR(w)
2   initialize P = N = ∅;
3   while(∃ab[♯(ab, w) ⩾ 2]) do{
4     P ← repetition(w, N);          (replacing all repetitions)
5     P ← arrangement(w, N);         (replacing frequent pairs)
6   }
7   if(|w| = 1) return P;
8   else return P ∪ {S → w};
9 end.
```

notation: $X \leftarrow Y$ denotes the addition of the set Y to X.

Fig. 1. The approximation algorithm for Grammar-Based Compression. An input is a string and an output is a set of production rule of a deterministic CFG for w.

```
1 procedure repetition(w, N)
2   initialize P = ∅;
3   while(∃w[i, i + j] = a^+)do{
4     replace w[i, i + j] by A_(a,j);
5     P ← {A_(a,j) → B_(a,j)C_(a,j)} and N ← {A_(a,j), B_(a,j), C_(a,j)},
6       where B_(a,j), C_(a,j) and their productions are recursively defined below;
7   }
8   return P;
9 end.
```

$$B_{(a,j)}C_{(a,j)} = \begin{cases} A^2_{(a,j/2)}, & \text{if } j \geqslant 4 \text{ is even} \\ A_{(a,j-1)} \cdot a, & \text{if } j \geqslant 3 \text{ is odd} \\ a^2, & \text{otherwise} \end{cases}$$

Fig. 2. The procedure *repetition*(,). An input is a string and a current alphabet. An output is a set of production rules deriving all repetitions in the input.

any repetition $w[i, j] = a^+$ in the input string by an appropriate nonterminal. More precisely, if an input string contains a repetition $w[i, j] = a^k$, then $w[i, j]$ is replaced by a nonterminal $A_{(a,k)}$ and the production $A_{(a,k)} \to B_{(a,k)}C_{(a,k)}$ is defined. The nonterminals $B_{(a,k)}, C_{(a,k)}$ and their productions are also defined recursively depending on k; $B_{(a,k)} = C_{(a,k)} = A_{(a,k/2)}$ if k is even and $B_{(a,k)}A_{(a,k-1)}$ and $C_{(a,k)} = a$ otherwise. Here, we show an example run of *repetition*(,) in Example 1.

Example 1. Let us consider the sample string $-a^7-a^4-a^5-$, where $a \in \Sigma$ and each—is a symbol not equal to a. *repetition*(,) replaces all repetitions as follows. The first repetition a^7 is replaced by A_7 and a^7 is recursively parsed by the production rules $A_7 \to A_6a$, $A_6 \to A_3^2$, $A_3 \to A_2a$, and $A_2 \to a^2$. Similarly, a^4, a^5 are replaced by A_4, A_5 and $A_4 \to A_2^2$, $A_5 \to A_4a$ are additionally defined.

On the other hand, the task of *arrangement* is to decide whether the algorithm replace a segment $w[i, i + 1] = ab$ by a nonterminal for each pair $ab \in \Sigma^2$, where $a \neq b$. This process is executed in the frequent order of all pairs stored in a priority queue indicated by *list* in line 3 of Fig. 3. This order is fixed until all elements are popped according to the following process.

We next briefly explain the task of *arrangement*. The complete description and an example are shown in the next subsection. Taking a most frequent pair ab from the priority queue and a unique index $id^{ab} = \{d_1^{ab}, d_2^{ab}\}$ is set for ab, where the index is simply denoted by $id = \{d_1, d_2\}$ if it is not necessary to indicate the pair. Let S be the set of segments $w[i, i + 1]$ such that $w[i, i + 1] = ab$. The task is to assign either d_1 or d_2 to each $s \in S$. Such an index is used to decide the replacement of the adjoining segment of s. Similarly, the replacement of s itself is decided by the index of its adjoining segment, which is already assigned. After the set $S' \subseteq S$ of segments to be replaced is decided, *arrangement* creates an appropriate nonterminal A and the production $A \to ab$.

After all pairs are popped from the priority queue, the algorithm actually replaces all the segments by their corresponding nonterminals. The obtained string w is given to the algo-

```
1   procedure arrangement(w, N)
2     initialize D = ∅;
3     make list: the frequency list of all pairs in w;
4     while(list is not empty)do{
5       pop the top pair ab in list;
6       set the unique id = {d1, d2} for ab;
7       compute the following sets based on C_ab = {w[i, i + 1] = ab}:
8         F_ab = {s ∈ C_ab | s is free},
9         L_ab = {s ∈ C_ab | s is left-fixed},
10        R_ab = {s ∈ C_ab | s is right-fixed};
11      D ← assignment(F_ab) ∪ assignment(L_ab) ∪ assignment(R_ab);
12    }
13    replace all segments in D by appropriate nonterminals;
14    return the set P of production rules computed by D and update N by P;
15  end.

16    subprocedure assignment(X)
17      in case(X = F_ab){D ← F_ab and set id(s) = d1 for all s ∈ F_ab};
18      in case(X = L_ab    (resp. X = R_ab))do{
19        compute the set Y of all left (resp. right) segments of X;
20        for each( yx ∈ YX    (resp. xy ∈ XY ))do{
21          in case (1): y is a member of an irregular subgroup,
22            set id(x) = d2;
23          in case (2): y is a member of an unselected subgroup,
24            set id(x) = d1 and D ← {x};
25          in case (3): y is a member of a selected subgroup,
26            if the group has an irregular subgroup,
27              set id(x) = d2;
28            else if the group has an unselected subgroup,
29              set id(x) = d1;
30            else if Y contains an irregular subgroup,
31              set id(x) = d2;
32            else set id(x) = d1;
33        }
34      }
35      return D;
36    end.
```

notation: $yx \in YX$ in line 20 denotes $y = w[i-1, i] \in Y$ and $x = w[i, i+1] \in X$.

Fig. 3. The procedure *arrangement*(,) and its subprocedure *assignment*( ). An input is a string and a current alphabet. The output is a set of production rules.

rithm as a next input and the two procedures are executed for w. The algorithm continues this process until there is no more pair ab such that $\sharp(ab, w) \geqslant 2$. In the next subsection we begin with the preparation of several notions to explain the details of *arrangement*(,).

3.2. Decision rule for assignment

In *arrangement*(,), the unique index $id^{ab} = \{d_1^{ab}, d_2^{ab}\}$ of integers is set for each pair ab, where this index is usually denoted by $id = \{d_1, d_2\}$ for the simplicity. All the segments

of ab are assigned either d_1 or d_2 as well as some of them are added to a current dictionary D. The indices assigned for the segments and the current D are the factors to decide the replacement of a segment which is not assigned any index yet. In this subsection we give the decision rule for the replacement of nonterminals.

Definition 1. A set of segments of a pair ab is called a *group* if all segments in the set are assigned by the index $id = \{d_1, d_2\}$ for ab. A group S is divided into at most two disjoint subsets S_1 and S_2 ($S_1 \cup S_2 = S$) such that S_1 is the set of segments assigned d_1 only and S_2 is defined similarly. S_1 and S_2 are said to be *subgroups* of the group S. Moreover subgroups of a group are categorized into the following three types depending on the current dictionary D. A subgroup is said to be *selected* if all segments in the subgroup are in D, *unselected* if all segments in the subgroup are not in D, and *irregular* otherwise.

Definition 2. A segment is called to be *free* if the left and right segments of it are not assigned, and is called to be *left-fixed* (*right-fixed*) if only the left (right) segment of it is assigned, respectively.

The assignment for segments are decided in the following manner. Let ab be a current pair popped from the priority queue. At first, the sets F_{ab}, L_{ab}, and R_{ab} are computed based on the set C_{ab} of all segments of ab. We then define $F_{ab} = \{s \in C_{ab} \mid s \text{ is free}\}$, $L_{ab} = \{s \in C_{ab} \mid s \text{ is left-fixed}\}$, and $R_{ab} = \{s \in C_{ab} \mid s \text{ is right-fixed}\}$.

The assignments for any segments $s \in F_{ab}$ and $s' \in C_{ab} \setminus F_{ab} \cup L_{ab} \cup R_{ab}$ are obviously decided. s is assigned d_1 and added to the current dictionary D. The assignment for s' is skipped and s' is never added to D.

The remained sets are L_{ab} and R_{ab}. We explain the case of L_{ab} only since the case of R_{ab} is symmetrically explained. L_{ab} is the set of the left-fixed segments, that is, the left side of each segment in L_{ab} is assigned and the other is not. Then let L be the set of the left segments of L_{ab}, that is, $L = \{w[i-1, i] \mid w[i, i+1] \in L_{ab}\}$.

All segment in L are assigned by the indices defined for some k pairs in Σ^2 ($k \geqslant 1$). Here we denote the indices by $id^1 = \{d_1^1, d_2^1\}, \ldots, id^k = \{d_1^k, d_2^k\}$. Thus, L is divided into k disjoint groups like $L = L_1 \cup L_2 \cup \cdots \cup L_k$ such that for each ℓ ($1 \leqslant \ell \leqslant k$), L_ℓ is assigned by $id^\ell = \{d_1^\ell, d_2^\ell\}$ and each group L_ℓ consists of at most two subgroups defined by id^ℓ.

Given such L_{ab} and L, the procedure *arrangement* finds all $w[i-1, i] \in L$ belonging to an unselected subgroup and then adds their all right segments $w[i, i+1] \in L_{ab}$ to D.

Next the assignments for L_{ab} are decided as follows. Each $w[i, i+1] \in L_{ab}$ is assigned d_2 if the left segment $w[i-1, i] \in L$ is in an irregular subgroup and each $w[i, i+1] \in L_{ab}$ is assigned d_1 if $w[i-1, i] \in L$ is in an unselected subgroup.

Any remained segment is $w[i, i+1] \in L_{ab}$ such that its left segment $w[i-1, i] \in L$ belongs to a selected subgroups of a group. In this case, the procedure checks whether the group also contains an unselected or irregular subgroup. If it contains an irregular subgroup, $w[i, i+1]$ is assigned d_2, else if it contains an unselected subgroup, $w[i, i+1]$ is assigned d_1, and otherwise, the procedure checks whether there is other group containing an irregular subgroup; If so, $w[i, i+1]$ is assigned d_2 and else $w[i, i+1]$ is assigned d_1. Consequently, a single group for L_{ab} assigned d_1 or d_2 is constructed from k groups $L = L_1 \cup L_2 \cup \cdots \cup L_k$.

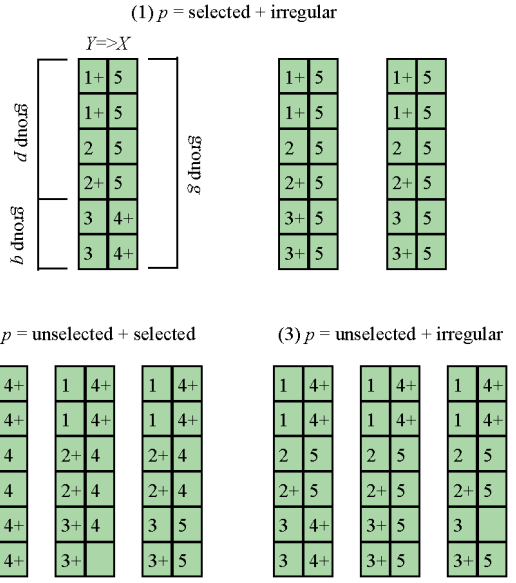

Fig. 4. Deciding the assignment for a set X of current segments from the set Y of their left segments and a current dictionary D.

Example 2. Fig. 4 illustrates how to decide the assignment for the set X of left-fixed segments. Y is the set of left segments of X. We assume that X is left-fixed, e.g., all segments in Y are already assigned and classified into the groups p and q. The group g for X is obtained from group p and q. The indices of group p, q, and g are denoted by $id = \{1, 2\},\ \{3\},\ \{4, 5\}$, respectively. The mark '+' denotes that the marked segments are added to a current dictionary D. On the first figure (1), since Y contains an *unselected* subgroup, then the corresponding segments in X are added to D. On the other hand, the assignments for X is decided as follows. There are an *irregular* subgroup and an *unselected* subgroup, then the corresponding segments in X are assigned the different indices 4 and 5, respectively. Finally, the remained segments are members of a *selected* subgroup. The case of a selected subgroup is synchronized with the assignment for a subgroup contained in the same group. In this case the selected subgroup is contained in group p. This group also contains the *irregular* subgroup. Thus, the assignment for the corresponding segments in X are assigned 5. Fig. 4 shows only the case that q consists of a single subgroup; nevertheless this figure is sufficiently general since the assignment for X is invariable even if q contains other subgroups.

3.3. Data structure

In this paper we assume two special data structures. One is the *doubly-linked list* to store an input string and the other is the *priority queue* for the frequency of all pairs. To gain constant-time access to ith occurrence of a pair ab from both $(i-1)$th and $(i+1)$th occurrence of it, we construct the doubly-linked list which was already implemented in [10] as follows.

A component of this doubly-list corresponds to a segment $w[i, i+1]$ of a pair $ab \in \Sigma^2$, where $1 \leqslant i \leqslant |w| - 1$. A component n_i consists of five members $w[i]$, *suc*(i), *pre*(i), *latter*(i), and *former*(i): *suc*(i) and *pre*(i) are pointers for n_{i-1} and n_{i+1}, respectively, *latter*(i) is the pointer for its successor, that is, the component n_{next} corresponding to the right-most segment of ab from $w[i, i+1]$ and *former*(i) is the pointer for its predecessor. Thus, each component n_i is constructed from the symbol $w[i]$ and four pointers. The construction time of this structure is $O(n)$ time.

On the other hand, the priority queue stores the frequency list of all pairs. This list is used in line 3 of the procedure *arrangement*. Initially, this list is empty. For each counting of a pair ab, we must renew this list in $O(1)$ time. In [10], this is realized by hash table but we introduce other data structure for the strict linear time compression.

Let $t : \Sigma^2 \to \mathbb{N} \times \mathbb{N}$ be a mapping form pairs ab to pairs of integers i, j. Let p and q be two arrays such that $p[i] = ab$ and $q[j] = k$ for some integers i, j, k and a pair ab. We call a triple (p, q, t) a *priority queue* and the semantics is defined as follows.

p is an array of all pairs in the frequent order and it is divided into $p_m \cdots p_1$ such that each p_j is the array of all pairs appearing j times. q is the array of the first members of all p_j for $1 \leqslant j \leqslant m$, that is, $q = p_1[0] \cdots p_m[0]$. We set $t(ab) = (i, j)$ iff $p[i] = ab$ and $q[j] = k$ for some integer k. This means that ab is the ith member in the current frequency list, the frequency of ab is j, and the position of the left-most pair in p with frequency j is k.

When the algorithm counts a new pair ab, using a temporary variable tmp, the priority queue is renewed in $O(1)$ time by the computation $tmp \leftarrow p[k]$, $p[k] \leftarrow p[i]$, $p[i] \leftarrow tmp$, $q[j] \leftarrow k+1$, and $t(ab) = (k, j+1)$. A mapping t is realized by an ordered tree in depth 2. Thus, the above priority queue for the frequency of all pairs can be constructed in $O(n)$ time and renewed in $O(1)$ time for each pair.

4. Approximation ratio and running time

In this section we prove the approximation ratio of our algorithm as well as its running time. We first show that an execution of the while-loop of the algorithm takes $O(n)$ time.

Lemma 1. *Each execution of the while-loop in algorithm* LEVELWISE-REPAIR *takes at most* $O(n)$ *time, where* n *is the length of an input string.*

Proof. By using a counter, for each repetition x^k in w, we can construct all nonterminals in the binary derivation for x^k in $O(k)$ time. Thus, the required time for *repetition*(w, N) is $O(n)$.

The doubly-linked list and priority queue defined in Section 3 are constructed in $O(n)$ time, respectively. Whenever the most frequent pair, say ab, is popped, the total length traced by the algorithm to compute the set C_{ab}, F_{ab}, L_{ab}, and R_{ab} is at most $O(k)$ for the number k of all occurrences of ab. Similarly, the sets L for F_{ab} and R for R_{ab} can be computed in $O(k) = O(n)$ time. By using an array of length at most n, we can preserve the subgroups for all segments $w[i, i+1]$ and get constant-time access to them. Thus, for each segment $w[i, i+1] \in L_{ab}$, we can decide the subgroup for $w[i-1, i] \in L$ in $O(1)$

time. The other conditions can be also computed in O(1) time. Hence, the running time of *arrangement*(,) for a pair ab is also O(n) time. Therefore, the time for any execution of the while-loop is bounded by O(n) time. □

Since a string output by *arrangement*(,) is shorter than its input (if not, the algorithm terminates), the number of execution of the outer-loop is at most n. Thus, the running time of *Levelwise-repair* is clearly bounded by O(n^2). We next show that the bound can be reduced to O(n) and the approximation ratio is bounded by O($\log \frac{n}{g_*}$).

Lemma 2. *Let w be an input string for repetition*(,), *$w[i_1, j_1] = w[i_2, j_2]$ be nonoverlapping intervals of a same substring in w, where $1 \leqslant i_1 < j_1 < i_2 < j_2 \leqslant |w|$. Let w' be the string output for w. Let I_1 be the shortest interval in w' satisfying that I_1 corresponds to an interval in w which contains $w[i_1, j_1]$. The other interval I_2 is similarly defined. Then it holds that $I_1[2, |I_1| - 1] = I_2[2, |I_2| - 1]$.*

Proof. We can assume $w[i_1, j_1] = w[i_2, j_2] = usv$ such that $u = a^+$ and $v = b^+$ for some $a, b \in N$. The intervals $w[i_1 + |u|, i_1 + |us| - 1] = w[i_2 + |u|, i_2 + |us| - 1] = s$ are compressed into a same string $\tilde{s}$. There exist $i \leqslant i_1$ and $i' \leqslant i_2$ such that $w[i, i_1] = w[i', i_2] = a^+$ are compressed into some symbols A_1 and A_2, and such indices exist also for j_1 and j_2. Thus, the strings represented by the intervals in w' corresponding to $w[i_1, j_1]$ and $w[i_2, j_2]$ are of the form $A_1\tilde{s}B_1$ and $A_2\tilde{s}B_2$, respectively. Hence these intervals satisfies the statement. □

Lemma 2 shows that any intervals represented by a same substring are compressed by *repetition*(,) into intervals which have a sufficiently long common substring. The main purpose of this section is to show that the same property is satisfied for the compressed strings by *arrangement*(,). We then prepare several notions prior to the proof.

Let p and q be pairs in a priority queue constructed in *arrangement*(,). A pair p is said to be *more frequent* than q if p is former element than q in the queue. Similarly, a segment s is also said to be *more frequent* than s' if the pair of s is more frequent than that of s'.

Definition 3. An interval $w[i, j]$ is said to be decreasing if $w[k, k+1]$ is more frequent than $w[k+1, k+2]$ for all k, $i \leqslant k \leqslant j-2$, and conversely, is said to be increasing if $w[k+1, k+2]$ is more frequent than $w[k, k+1]$ for all $i \leqslant k \leqslant j-2$. A segment $w[i, i+1]$ is said to be local maximum if $w[i, i+1]$ is more frequent than $w[i-1, i]$, $w[i+1, i+2]$ and is said to be local minimum if $w[i-1, i]$, $w[i+1, i+2]$ are more frequent than $w[i, i+1]$.

Here we note that any repetition like a^+ is replaced by a nonterminal. Thus for any two segments representing different strings, one of them is more frequent than the other.

Definition 4. Let $w[i, j]$ and $w[i', j']$ be independent occurrences of a substring and D be a current dictionary. Let s_k and s'_k be the kth segments in $w[i, j]$ and $w[i', j']$ from the left most segments, respectively. Then s_k, s'_k are said to agree with D if either $s_k, s'_k \in D$ or $s_k, s'_k \notin D$, and are said to disagree with D otherwise.

Lemma 3. *Let w be an input for arrangement$(\,,\,)$, $w[i, i+j] = w[i', i'+j]$ be independent occurrences of a same substring in w, and D be the computed dictionary. Then the following two conditions hold*: (1) *for each k, $6 \leqslant k \leqslant j-6$, two segments $w[i+k, i+k+1]$ and $w[i'+k, i'+k+1]$ agree with D and* (2) *for each intervals $w[\ell, \ell+3]$ contained in $w[i, i+j]$, at least one segment in $w[\ell, \ell+3]$ is in D.*

Proof. We first show condition (1). If $w[i, i+j]$ contains a local maximum segment $s = w[i+k, i+k+1]$, then s is the first segment chosen from $w[i+k-1, i+k+2]$. Thus, s and the corresponding segment s' in $w[i', i'+j]$ are added to D and assigned a same index.

Similarly it is easy to see that any segments agree with D between two adjacent local maximum segments. Thus, the remained intervals are a long decreasing prefix and a long increasing suffix of $w[i, i+j]$ and $w[i', i'+j]$. In order to prove this case, we need the following claim:

Claim 1. *Let id be the index for a pair in Σ^2. Any group defined by id contains at most two different subgroups in selected, unselected, and irregular.*

This claim is directly obtained from Definition 1 (see Fig. 4 which illustrates all the cases). Let $w[i, i+j]$ contains a decreasing prefix of length at least six. The first segment chosen from the prefix of $w[i, i+j]$ is $w[i, i+1]$, and $w[i', i'+1]$ is also chosen simultaneously. They are then classified into some groups. Since the prefix is decreasing, succeedingly chosen segments are the right segments of $w[i, i+1]$ and of $w[i', i'+1]$. They are indicated by s and s', respectively. Since s and s' are both left-fixed and represent a same pair, they are classified into a same group g.

Case 1: The group g consists of a single subgroup. In this case, s and s' are both contained in one of (a) selected, (b) unselected, and (c) irregular subgroup. Case (a) satisfies that s and s' are assigned a same index and are both added to D. Thus, from the segments, no disagreement happens within the prefix. Case (b) and (c) converge to (a) within at least two right segments from s are chosen.

Case 2: The group g containing s and s' consists of two different subgroups. The right segments of s and s' are assigned some indices according to the types of the groups in which s and s' are contained. All the combinations of two different subgroups are (i) selected and unselected, (ii) selected and irregular, and (iii) unselected and irregular. In the first two cases, the right segments are all classified into a single subgroup. In the last case, any segment are classified into a selected or unselected subgroup, that is, this case converges to case (i). Thus, all case of (i), (ii), and (iii) converge to case 1 within further two right segment from s and s' are chosen.

Consequently, it is guaranteed that two segments $w[i+k, i+k+1]$ and $w[i'+k, i'+k+1]$ are assigned a same index and they are added to D within four right segments from s and s' are chosen. It follows that any disagreement of $w[i, i+j]$ and $w[i', i'+j]$ in the decreasing prefix happens within only the range $w[i, i+6]$ and $w[i', i'+6]$. The increasing suffix case can be similarly shown.

We next show condition (2). Since all local maximum segments are added to D, the possibility for unsatisfying condition (2) is only the cases of a decreasing prefix and increasing

suffix of $w[i, i+j]$. As is already shown in the above, any segment is classified into one of a selected, unselected, and irregular subgroup. Moreover, the last two subgroups converge to a selected subgroup within two segments. Thus, $w[i, i+j]$ and $w[i', i'+j]$ has no three consecutive segments which are not added to D. □

Theorem 1. *The running time of* LEVELWISE-REPAIR *is bounded by* $O(n)$ *for each input string of length* n.

Proof. By Lemma 3, we obtain $|w'| \leqslant \frac{3}{4}|w|$ for each input string w and its compressed string w' by one execution of the while-loop. Thus, by Lemma 1, the total number of symbols accessed by the algorithm is bounded by $O(n)$. □

Example 3. Fig. 5 illustrates the convergence of a long prefix case. Let a string w contain 8 independent intervals which have the same prefix '*abcdefg*' and this prefix be decreasing. The 1–8 rows represent such 8 intervals. Assume that the set of segments of ab are already assigned and classified into two group p and q. The last 4 rows correspond to other 4 intervals in w which have the same prefix '*bcdefg*'. This figure shows that the assignments for all 12 rows converge on the column of cd in a same group. All segments of this group are set to a same *selected* subgroup on this column. We note that the convergence of 1–8 rows is guaranteed regardless of the last 4 rows since for each group g', the assignments for right segments of g' are not affected by other groups as long as g' contains 2 subgroups (see Example 2). Finally each interval is compressed in the string shown in its right side.

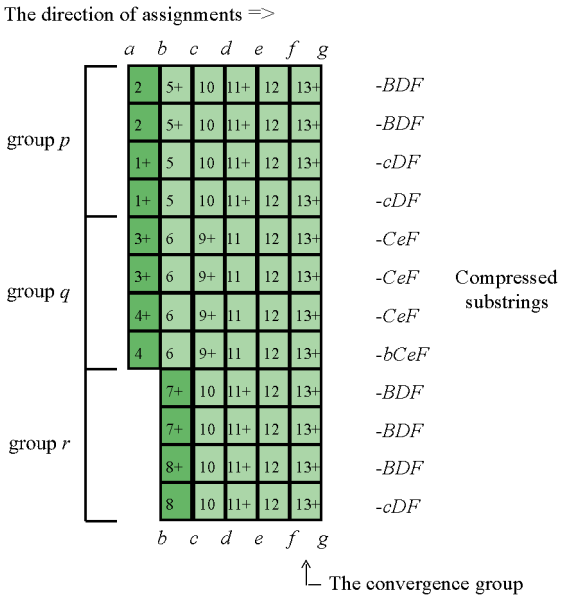

Fig. 5. The convergence of assignments for a long *decreasing* prefix case.

Nonterminals B, C, D, E, and F are associated with the production rules $B \to bc$, $C \to cd$, $D \to de$, $E \to ef$, and $F \to fg$, respectively. The left most '$-$' denotes an indefinite character since they depend on their left sides.

Finally, we show the main result of this paper by comparing the size of output grammar G with the *LZ-factorization* [20] of w. Here we recall its definition: the LZ-factorization of w denoted by $LZ(w)$ is the decomposition $w = f_1 \cdots f_k$, where $f_1 = w[1]$ and for each $1 \leqslant \ell \leqslant k$, f_ℓ is the longest prefix of $f_\ell \cdots f_k$ which occurs in $f_1 \cdots f_{\ell-1}$. Each f_ℓ is called a *factor*. The size of $LZ(w)$, denoted by $|LZ(w)|$, is the number of its factors.

Theorem 2 [15]. *For each string w and its minimum deterministic CFG G, it holds that $|LZ(w)| \leqslant |G|$.*

Theorem 3. *The approximation ratio of* LEVELWISE-REPAIR *is* $O(\log \frac{n}{g_*})$ *for every string of length n and the size g_* of its minimum grammar-based compression.*

Proof. By Theorem 2, it is sufficient to prove $|G|/|LZ(w)| = O(\log \frac{n}{g_*})$. We begin proving the following fact.

Fact 1. *Let $w = f_1 \cdots f_k$ be the LZ-factorization. For each i, $2 \leqslant i \leqslant k$, either f_i appears also in $f_1 \cdots f_{i-1}$ or $|f_i| = 1$.*

Let $\#(w)$ be the number of new nonterminals produced by single execution of the while-loop for a string w. From Fact 1 and Lemma 3, if $|f_i| \geqslant 2$, then it appears in w at least twice and such two occurrences are compressed into the almost same substrings $\alpha\beta\gamma$ and $\alpha'\beta\gamma'$ such that $|\alpha\gamma\alpha'\gamma'|$ is bounded by a constant. Thus, we obtain the equation $\#(w) = \#(f_1 \cdots f_{k-1}) + O(1) = \#(f_1 \cdots f_{k-2}) + O(1) + O(1) = \cdots = O(k) = O(g_*)$. Hence, the number of produced new nonterminals is bounded by $O(g_*)$.

Since $|w'| \leqslant \frac{3}{4}|w|$ for a string w and its compressed string w' by one execution of the while-loop for w, the number of executed while-loops is bounded by $O(\log n)$. It follows that $|G|/|LZ(w)| = O(\log n)$. This ratio can be improved to $O(\log \frac{n}{g_*})$ by the following careful estimation for the depth of the while-loop which contributes to the increase of new nonterminals.

For each factor f_i, let N_i be the set of produced nonterminals for f_i by the time f_i is compressed to the final string. Let $X_i = \{A \in N_i \mid \exists j < i\ [A \in N_j]\}$ and $Y_i = N_i \setminus X_i$. Using Lemma 3 again, the number $|N|$ of total nonterminals is estimated by

$$|N| \leqslant \sum_{i=1}^{k} \bigl(|X_i \cup Y_i|\bigr) + k = \sum_{i=1}^{k} |Y_i| + k = \sum_{i=1}^{k} c_i \log |f_i| + k$$

for some constants c_i.

Since $|f_1| + \cdots + |f_k| = n$, $\sum_{i=1}^{k} c_i \log |f_i|$ is maximum in case $|f_i| = n/k$ for each $1 \leqslant i \leqslant k$. Thus, $|N| \leqslant ck \log \frac{n}{k} + k$, where $c = \max\{c_1, \ldots, c_k\}$. Therefore, we obtain $|G|/|LZ(w)| = O(|N|/|LZ(w)|) = O(\log \frac{n}{g_*})$. □

5. Conclusion

For the grammar-based compression with constant alphabets, we presented a fully linear-time approximation algorithm. This algorithm guarantees the best known approximation ratio $O(\log \frac{n}{g_*})$ without suffix tree construction. Since a suffix tree requires over $10n$ space for the length n of an input string, it is difficult to implement compression algorithms using suffix tree [4,5]. On the other hand, our algorithm requires only the data structures used by the practical algorithm RE-PAIR in [10]. The space required by RE-PAIR is $5n + 4k^2 + 4k' + \lceil\sqrt{n}\rceil$ space, the size of initial alphabet k, and the size of final alphabet k'. In our algorithm, it is shown that $k' = O(g_* \log \frac{n}{g_*})$. Thus, the space required by our algorithm is approximately $5n + \lceil\sqrt{n}\rceil + O(g_* \log \frac{n}{g_*})$. This is usually considered to be smaller than $10n$. Actually, the space efficiency of RE-PAIR is reported in [10] by several experiments.

There are several open problems. An important problem is an upper bound of the approximation ratio of RE-PAIR algorithm [10]. Other problem is to construct a space-economic approximation algorithm preserving the approximation ratio.

References

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

[2] S. De Agostino, J.A. Storer, On-line versus off-line computation in dynamic text compression, Inform. Process. Lett. 59 (1996) 169–174.

[3] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, A. Shelat, Approximating the smallest grammar: Kolmogorov complexity in natural models, in: Proc. 29th Ann. Symp. Theory Comput., 2002, pp. 792–801.

[4] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. 38th Ann. Symp. Found. Comput. Sci., 1997, pp. 137–143.

[5] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Computer Science and Computational Biology, Cambridge University Press, 1997.

[6] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa, Collage system: a unifying framework for compressed pattern matching, Theoret. Comput. Sci., submitted for publication.

[7] J.C. Kieffer, E.-H. Yang, Grammar-based codes: a new class of universal lossless source codes, IEEE Trans. Inform. Theory 46 (3) (2000) 737–754.

[8] J.C. Kieffer, E.-H. Yang, G. Nelson, P. Cosman, Universal lossless compression via multilevel pattern matching, IEEE Trans. Inform. Theory 46 (4) (2000) 1227–1245.

[9] D. Knuth, Seminumerical Algorithms, Addison-Wesley, 1981, pp. 441–462.

[10] N.J. Larsson, A. Moffat, Offline dictionary-based compression, Proc. IEEE 88 (11) (2000) 1722–1732.

[11] E. Lehman, Approximation algorithms for grammar-based compression, PhD Thesis, MIT, 2002.

[12] E. Lehman, A. Shelat, Approximation algorithms for grammar-based compression, in: Proc. 20th Ann. ACM-SIAM Symp. Discrete Algorithms, 2002, pp. 205–212.

[13] C. Nevill-Manning, I. Witten, Compression and explanation using hierarchical grammars, Computer J. 40 (23) (1997) 103–116.

[14] C. Nevill-Manning, I. Witten, Identifying hierarchical structure in sequences: a linear-time algorithm, J. Artificial Intelligence Res. 7 (1997) 67–82.

[15] W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, in: Proc. 13th Ann. Sympo. Combinatorial Pattern Matching, 2002, pp. 20–31.

[16] H. Sakamoto, A fully linear-time approximation algorithm for grammar-based compression, in: Proc. 14th Ann. Symp. Combinatorial Pattern Matching, 2003, pp. 348–360.

[17] J.A. Storer, T.G. Szymanski, The macromodel for data compression, in: Proc. 10th Ann. Symp. Theory Comput., ACM Press, San Diego, CA, 1978, pp. 30–39.
[18] T.A. Welch, A technique for high performance data compression, IEEE Comput. 17 (1984) 8–19.
[19] E.-H. Yang, 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) (2000) 755–777.
[20] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–349.
[21] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.