A Fully Linear-Time Approximation Algorithm for Grammar-Based Compression

Sakamoto, Hiroshi
Department of Informatics, Kyushu University

出版情報：DOI Technical Report. 214, 2003-01. Department of Informatics, Kyushu University

A Fully Linear-Time Approximation Algorithm for Grammar-Based Compression

Hiroshi Sakamoto*
hiroshi@i.kyushu-u.ac.jp

Abstract

A linear-time approximation algorithm for the *grammar-based compression*, which is an optimization problem to minimize the size of a context-free grammar deriving a given string, is presented. Given a string of length n, the algorithm guarantees $O(\log^2 n)$ approximation ratio and using the data structures of doubly-linked list, hash table, and priority queue, it runs in $O(n)$ time even if the size of alphabet is unbounded.

1 Introduction

The grammar-based compression is an optimization problem, given an input string, to find a small context-free grammar which generates the single string. This problem is known to be NP-hard and not approximable within a constant factor [9], and due to a relation with an algebraic problem [6], it is unlikely to found an algorithm approximating this problem within $O(\log n/\log\log n)$.

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 [16] (including LZW [13]) and BISECTION [5] encodings are considered as algorithms to find a straight-line program, which is a very restricted CFG. Lehman and Shelat [9] also showed the lower bounds of the approximation ratio of almost dictionary-based encodings to the smallest CFG, and unfortunately, these lower bounds are relatively large to $O(\log n)$ ratio.

The first polynomial-time algorithms which guarantee a *small* approximation ratio were produced by Charikar, Lehman, Liu, et al. [1], and Rytter [12], independently. In particular, the latter algorithm is attracted by the simplicity of the algorithm in the view point of its implementation for large text data.

Rytter's algorithm runs in $O(n \log |\Sigma|)$ time for unbounded alphabet Σ and in linear time for any constant alphabet. This gap is caused by the construction of a suffix tree in the algorithm to retrieve whether a string appear in the input in linear time. The edges labeled by characters leaving a node are lexicographically sorted. Thus, in this representation, sorting is a lower bound for suffix tree construction and one open problem remains whether there is a linear-time polylog-approximation algorithm for the grammar-based compression even in case of unbounded alphabet.

The starting point of this study is RE-PAIR encoding by Larsson and Moffat [7] which recursively replaces all pairs like ab in an input string according to the frequency. This

*Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan

encoding scheme 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 [8]. Thus, a nontrivial upper bound of the approximation ratio of RE-PAIR is still an important open problem.

Our algorithm is not RE-PAIR in itself but is based on the strategy of the recursive replacement of pairs. Consider a situation that a string contains nonoverlapping intervals X and Y which represent a same substring. The aim of our algorithm is to compress X and Y into some intervals which have a common substring as long as possible. More precisely, X and Y are aimed to be compressed into $X' = \alpha\beta\gamma$ and $Y' = \alpha'\beta\gamma'$ so that the length of the total disagreement $\alpha\gamma$ is bounded by a constant. If this encoding is realized for all such 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 that X and Y are partitioned by some delimiter characters on their both sides, it is easy to compress them into an same string by RE-PAIR strategy. However X (or Y) is generally overlapping with other intervals which represent other different substrings. The main goal of this paper is that our algorithm executes the required encoding in general case without suffix tree.

We call our algorithm LEVELWISE since the replacement of pairs is restricted by the level in which the pairs exist, that is, once an interval is replace by a nonterminal, any interval containing it is not replaced within the same loop.

In this paper, we assume a standard RAM model for reading any $O(\log n)$ bit integer in constant-time. We additionally assume three data structures, *doubly-linked list*, *hash table*, and *priority queue* to gain constant-time access to any occurrence of a pair ab. The construction of such data structures for input string is presented in [7]. Using these structures, the running time of LEVELWISE is reduced to linear-time for unbounded alphabet.

The approximation ratio of LEVELWISE is obtained from the comparison with the size of the output grammar and the size of the *LZ-factorization* [15] for an input string. Since a logarithmic relation between LZ-factorizations and minimum grammars is already shown in [12], we can conclude a polylogarithmic approximation ratio of our algorithm.

2 Definitions

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

The ith symbol of w is denoted by $w[i]$ and $w[i,j]$ denotes the interval from $w[i]$ to $w[j]$. 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. A prefix α of β is called *proper* if $|\alpha| < |\beta|$ and a *proper suffix* is similar.

A substring $w[i,j] = x^k$ 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 have no need to specify the length k. Intervals $w[i,j]$ and $w[i',j']$ $(i < i')$ are called to be *overlapping* if $i' \leq j < j'$ and to be *independent* if $j < i'$. A substring ab of length two in a string w is called a *pair* in w. Similarly, an interval $w[i,i+1]$ is called a *segment* of ab if $w[i,i+1] = ab$. 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 $w[i,j]$, respectively.

A *context-free grammar* (CFG) is a 4-tuple $G = (\Sigma, N, P, S)$, where Σ and N are alphabets disjoint each other, P is a set of relations, called *production rules*, between N and strings over $\Sigma \cup N$, and $S \in N$ is called the *start symbol*. Elements in N are called *nonterminal*. A production rule in P represents a replacement rule, which is 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 grammar 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 $L(G)$ is defined by G is a singleton set, i.e., $|L(G)| = 1$.

The *size* of G, denoted by $|G|$, is the total length of right sides of all production rules. In particular, $|G| = 2|N|$ in case of Chomsky normal form. 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

3 An Approximation Algorithm

We present the approximation algorithm, named by LEVELWISE, for the grammar-based compression in Fig 1. This algorithm calls two procedures *repetition* and *assort* presented in Fig 2 and 3, respectively. We begin with the outline of the algorithm as well as the procedures below.

Outline of the algorithm: The *repetition* receives a string w and replaces all repetitions $w[i,j] = x^+$ of length k in w by a nonterminal $A_{(x,k)}$. A production $A_{(x,k)} \to BC$ is then added to P and nonterminals B, C are defined recursively such that $B = C = A_{(x,k/2)}$ provided k is even, and $BC = A_{(x,k-1)}x$ otherwise. Thus, the interval $w[i,j]$ is compressed by a nonterminal which is the root of a binary derivation tree of depth at most $O(\log k)$.

Next the *assort* receives w and counts the frequency of all pairs in w. All such pairs are managed by a priority queue in the frequent order, where two different pairs in a same frequency are ordered by FIFO manner. This queue is indicated by *list* in line 3 of Fig 3 and this order is fixed until all elements are popped as follows.

In the process of *assort*, a dictionary D is initialized and a unique index $id = \{d_1, d_2\}$ is created for each pair ab. The aim of the procedure is, for each segment $w[i, i+1] = ab$, to decide whether $w[i, i+1]$ is added to D and assign d_1 or d_2 to $w[i, i+1]$ by a decision rule. All segments in D are finally replaced by appropriate nonterminals.

After all pairs are popped from the priority queue, the algorithm actually replaces all segments in D; If $w[i, i+1] = w[i', i'+1] = ab$ and they are in D, then they are replaced a same nonterminal. The resulting string is then given to *repetition* as a next input and the algorithm continue this process until there is no more pair ab appearing in w at least twice.

In order to explain the decision rule evaluated in *assort* we introduce the following notions.

Definition 1 *A set of segments of a pair ab is called a group if all segments are assigned by the index $id = \{d_1, d_2\}$ for ab. A group consists of at most two disjoint subsets S_1 and S_2 assigned d_1 and d_2, respectively. Such subsets are said to be subgroups of the group. 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.*

Algorithm LEVELWISE(w)
  initialize $P = N = \emptyset$;
  while($\exists ab[\sharp(ab, w) \geq 2]$) **do**{
    $P \leftarrow$ *repetition*(w, N);   (*replacing all repetitions*)
    $P \leftarrow$ *assort*(w, N);   (*replacing frequent pairs*)
  }
  if$(|w| = 1)$ return P;
  else return $P \cup \{S \rightarrow w\}$;
end.

notation: $X \leftarrow Y$ denotes all members in Y are added to X.

Figure 1: The algorithm LEVELWISE. An input is a string and an output is a set of production rule of an admissible grammar for w.

procedure *repetition*(w, N)
  initialize $P = \emptyset$;
  while($\exists w[i, i+j] = a^+$)**do**{
    replace $w[i, i+j]$ by $A_{(a,j)}$;
    $P \leftarrow \{A_{(a,j)} \rightarrow BC\}$ and $N \leftarrow \{A_{(a,j)}, B, C\}$ recursively;
  }
  return P;
end.

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

Figure 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.

1 **procedure** $assort(w, N)$
2 initialize $D = \emptyset$;
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 $id = \{d_1, d_2\}$ for ab;
7 compute the following sets based on $C_{ab} = \{w[i, i+1] = ab\}$:
8 $F_{ab} = \{s \in C_{ab} \mid s \text{ is free }\}$,
9 $L_{ab} = \{s \in C_{ab} \mid s \text{ is left-fixed }\}$,
10 $R_{ab} = \{s \in C_{ab} \mid s \text{ is right-fixed }\}$;
11 $D \leftarrow assign(F_{ab}) \cup assign(L_{ab}) \cup assign(R_{ab})$;
12 }
13 replace all segments in D by appropriate nonterminals;
14 return the set P of production rules corresponding to D and update N by P;
15 **end.**

16 **subprocedure** $assign(X)$
17 **in case**($X = F_{ab}$){ $D \leftarrow F_{ab}$ and set $id(s) = d_1$ for all $s \in F_{ab}$;}
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**($yx \in YX$ (*resp.* $xy \in XY$))**do**{
21 in case (1): y is a member of an irregular subgroup,
22 set $id(x) = d_2$;
23 in case (2): y is a member of an unselected subgroup,
24 set $id(x) = d_1$ and $D \leftarrow \{x\}$;
25 in case (3): y is a member of a selected subgroup,
26 if the group has an irregular subgroup,
27 set $id(x) = d_2$;
28 else if the group has an unselected subgroup,
29 set $id(x) = d_1$;
30 else if Y contains an irregular subgroup,
31 set $id(x) = d_2$;
32 else set $id(x) = d_1$;
33 }
34 }
35 return D;
36 **end.**

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

Figure 3: The procedure *assort* and *assign*. An input is a string and a current alphabet. The output is a set of production rules which is selected by the frequency of pairs in the input string as well as by the levelwise strategy.

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

Decision rule for assignment: The assignment for segments are decided by *assort* as the following manner. Let ab be a current pair popped from the priority queue. At first, the sets F_{ab}, L_{ab}, R_{ab}, and C'_{ab} are computed based on the set C_{ab} of all segments $w[i, i+1] = ab$.

F_{ab} is the set of free segments, that is, the both sides of each $w[i, i+1] \in F_{ab}$ are not assigned. For each segment in F_{ab}, *assort* assigns the index d_1 and add it to the dictionary D. All segments registered to D are collectively replaced after the process of *assort* is finished.

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. Let L be the set of such assigned segments. *assort* decides the assignments for all $w[i, i+1] \in L_{ab}$ as well as whether $w[i, i+1]$ is added to D depending on L. The decision is evaluated as follows[1].

Since all segments in L are assigned, L is divided into some disjoint groups like $L = L_1 \cup L_2 \cdots \cup L_k$ such that L_ℓ is assigned by a unique $id = \{d_1, d_2\}$ and each group L_ℓ consists of some subgroups.

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

Next it decides the assignment for L_{ab} as follows. Assign d_2 to each $w[i, i+1] \in L_{ab}$ if the left segment $w[i-1, i] \in L$ is in an irregular subgroup and d_1 to each $w[i, i+1] \in L_{ab}$ if $w[i-1, i] \in L$ is in an unselected subgroup.

The remained segments are $w[i, i+1] \in L_{ab}$ such that the corresponding $w[i-1, i] \in L$ belong to a selected subgroup of a group. In this case, the procedure checks whether the group contains other subgroups, that is, unselected or irregular. If the group 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 in 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 $L = L_1 \cup L_2 \cdots \cup L_k$. The resulting group is used for further assignment of right segments of L_{ab}.

The case R_{ab} is symmetric, that is, the set R of the right assigned segments for R_{ab} is computed and the assignment and dictionary for R_{ab} are decided by R. The remained segments in $C'_{ab} = C_{ab} \setminus F_{ab} \cup L_{ab} \cup R_{ab}$ are skipped since both sides of any segments in C'_{ab} are already assigned.

We first show that the running time of our algorithm is in at most $O(n^2)$. This order is reduced to a linear time at the next section.

Proposition 1 LEVELWISE runs in at most $O(n^2)$ in the length of an input string.

proof. 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 $O(n)$. For other computation, we initially construct a doubly-linked list for w to gain

[1] This process and all concrete assignments are illustrated in Appendix **A**

constant-time access to any occurrence of a pair ab in w. Since this technique was already implemented in [7], we briefly explain the idea.

The length of the linked-list, that is the number of nodes is n such that the ith node n_i contains at most five pointers $a(i)$, $suc(i)$, $pre(i)$, $latter(i)$, and $former(i)$, where $a(i)$ is $w(i)$, $suc(i)$ and $pre(i)$ are pointers for the nodes n_{i-1} and n_{i+1}, respectively, $latter(i)$ is the pointer for the next occurrence of ab for $w[i, i+1] = ab$, and the $former(i)$ is similar. The time to construct this linked-list is $O(n)$.

The priority list of all pairs in w is simultaneously constructed. Whenever the top of the priority list, 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 $O(k)$ time.

Using hash table, for each $w[i, i+1] \in L_{ab}$ we can decide the group of the $w[i-1, i] \in L$ in $O(1)$ time. Moreover, other conditions can be also computed in $O(1)$ time. Thus, the running time of *assort* for a pair ab is also in $O(k)$. Since an output string by *assort* is shorter than its input (if not, the algorithm terminates), the number of repetitions of the outer-loop is at most n. Therefore, the running time of *Levelwise* is at most $O(n^2)$. □

4 Approximation Ratio and Running Time

In the section, we show that LEVELWISE is $O(\log^2 n)$-approximation algorithm as well as it runs in linear time in an input length. We first show that *repetition* compresses two independent intervals of a same substring into a sufficiently long common string.

Lemma 1 Let w be an input string for *repetition* and $w[i_1, j_1] = w[i_2, j_2]$ be nonoverlapping intervals of a same substring in w. Let w' be the resulting string and let I_1 and I_2 be two intervals in w' corresponding to $w[i_1, j_1]$ and $w[i_2, j_2]$, respectively. Then it holds that $I_1[2, |k|-1] = I_2[2, |k|-1]$, where k is the length of I_1.

proof. We can assume $w[i_1, j_1] = w[i_1, j_1] = usv$ such that $u = a^+$ and $v = b^+$ for some $a, b \in N$. The substrings $w[i_1+|u|, i_1+|us|-1] = w[i_2+|u|, i_2+|us|-1] = s$ are compressed into a same string $\tilde{s}$. There exist $i \leq i_1$ and $i' \leq i_2$ such that $w[i, i_1] = w[i', i_2] = a^+$ are compressed into a symbol A_1 and A_2, and such indices exist also for j_1 and j_2. Thus, the interval 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. They are the same string except one symbols of both sides of them. □

Let p and q be pairs in a priority queue constructed in *assort*. We define the partial order $\succ$ for all pairs such that $p \succ q$ if p is former element than q in the queue. Then, p is said to be *more frequent* than q. Particularly, the top element of a current queue is said to be the *most frequent* pair. Similarly, we write $w[i, i+1] \succ w[i', i'+1]$ if $w[i, i+1] = ab, w[i', i'+1] = a'b'$, and $ab \succ a'b'$.

Definition 3 *An interval $w[i, j]$ is said to be decreasing if $w[k, k+1] \succ w[k+1, k+2]$ for all $i \leq k \leq j-2$, and conversely, is said to be increasing if $w[k, k+1] \prec w[k+1, k+2]$ for all pairs. A segment $w[i, i+1]$ is said to be local maximum if $w[i, i+1] \succ w[i-1, i], w[i+1, i+2]$ and said to be local minimum if $w[i, i+1] \prec w[i-1, i], w[i+1, i+2]$.*

Here we note that any repetition like a^+ is replaced by a nonterminal by the first procedure *repetition*, any input given to *assort* contains no segment $w[i, i+2]$ satisfying $w[i, i+1] = w[i+1, i+2]$, that is, any different segments satisfy $w[i, i+1] \succ w[i', i'+1]$.

Definition 4 *Let $w[i,j]$ and $w[i',j']$ be independent occurrences of a substring and D be a dictionary, that is, a set of segments in w. Let s_k and s'_k be the kth segments from the $w[i,i+1]$ and $w[i',i'+1]$, respectively. Then, the segments s_k, s'_k are said to agree with D if $s_k, s'_k \in D$ or $s_k, s'_k \notin D$, and are said to disagree with D otherwise.*

Lemma 2 Let w be an input for *assort*, $w[i,i+j] = w[i',i'+j]$ be two independent occurrences of a same substring in w, and D be a dictionary computed by *assort*. Then, the following two conditions hold: (1) the segments $w[i+k,i+k+1]$ and $w[i'+k,i'+k+1]$ agree with D for any $6 \leq k \leq j-6$ and (2) $w[i,i+j]$ contains no interval $w[\ell,\ell+3]$ whose three segments are not in D.

proof. *proof of condition (1)*: If $w[i,i+j]$ contains a local maximum segment $s_1 = w[i+k,i+k+1]$, then s_1 is the first segment chosen from $w[i+k-1,i+k], s_1, w[i+k+1,i+k+2]$. Thus, s_1 and the corresponding segment s'_1 in $w[i',i'+j]$ are added to D and assigned a same index.

Similarly it is easy to see that any segments $w[i+k,i+k+1]$ and $w[i'+k,i'+k+1]$ agree with D between the left most and right most local maximum segments in $w[i,i+j]$ and $w[i',i'+j]$. Thus, the remained intervals are a long decreasing prefix and a long increasing suffix[2] of $w[i,i+j]$ and $w[i',i'+j]$. In order to prove this case, we need the following claims:

***claim* 1** Any group computed by *assort* consists of at most two different subgroups of selected, unselected, and irregular.

***claim* 2** When a segment s is chosen by *assort* to assign some index, if the left segment of s belongs to a group containing two different subgroups, then the assignment for s is decided by only the subgroups.

Claim 1 is directly obtained from Definition 1. Claim 2 is derived from the subprocedure *assign* (Appendix **A** shows all cases of assignments for such s). Let $w[i,i+j]$ contains a decreasing prefix of length at least six. The segment firstly 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 s of $w[i,i+1]$ and s' of $w[i',i'+1]$. 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, or (c) irregular subgroup. The 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. The case (b) and (c) converge to the case (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. By Claim 2, the right segments of s and s' are assigned by only the condition of this group. The all 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, each case of (i), (ii), and (iii) converges to Case 1 within further two right segment from s are chosen.

Consequently, together with Case 1 and 2, it is satisfied that some 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

[2]The decreasing prefix case is demonstrated in Appendix **B**.

right segment 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 case of an increasing suffix of them can be similarly shown.

proof of condition (2): Since all local maximum segments are added to D, the possibility for unsatisfying Condition (2) is remained only on 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, and the last two subgroups must 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. □

Finally, we show the main result of this paper by comparing the size of output grammar G with the *LZ-factorization* [15] 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 $1 \leq \ell \leq k$, 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 1 ([12]) For each string w and its deterministic CFG G, $|LZ(w)| \leq |G|$.

Theorem 2 For each string w of length n, the approximation ratio of LEVELWISE is $O(\log^2 n)$ ant it runs in $O(n)$.

proof. By Theorem 1, it is sufficient to prove $|G|/|LZ(w)| = O(\log^2 n)$. For each factor f_ℓ, the prefix $f_1 \cdots f_{\ell-1}$ contains at least one occurrence of f_ℓ. We denote f_ℓ by $w[i, i+j]$ and other occurrence by $w[i', i'+j]$, respectively. By Lemma 1 and 2, after one loop of the algorithm is executed, the substrings represented by $w[i, i+j]$ and $w[i', i'+j]$ are compressed into some strings $\alpha\beta\gamma$ and $\alpha'\beta\gamma'$, respectively, where $|\alpha|, |\gamma| \leq 4$. By Lemma 2, $|\beta| \leq \frac{3}{4}j$. Since β occurs in the compressed string at least twice, we can apply Lemma 1 and 2 to the strings until they are compressed into sufficiently short strings.

Thus, the interval $w[i, i+j]$ corresponding to f_ℓ is compressed into a string of length at most $O(\log j)$. It follows that w compressed into a string of length at most $O(k \log n)$, where $k = |LZ(w)|$. Hence, we can estimate $|G| = 2|N| + c \cdot k \log n$ with a constant c and the set N of all nonterminals of G.

The number of different nonterminals in the compressed string is at most $c \cdot k \log n$. If $A \in N$ occurs in the string and $A \rightarrow BC \in P$, then the pair BC must occur in the lower string at least twice. Thus, the number of different nonterminals in the lower level is also at most $c \cdot k \log n$. Since the depth of the loop of the algorithm is $O(\log n)$, $|N| \leq ck \log n \cdot \log n$ Hence, we obtain $|G|/|LZ(w)| = O(\log^2 n) + O(\log n) = O(\log^2 n)$.

The running time can be reduced in linear time in n since the number of repetitions of the outer loop of the algorithm is $O(\log n)$ and $|\beta| \leq \frac{3}{4} \cdot j$. □

5 Conclusion

For the grammar-based compression problem, we presented a fully linear time algorithm which guarantees $O(\log^2 n)$ approximation ratio for input strings over possibly unbounded alphabets. The remained open problem is whether this ratio can be reduced to $O(\log n)$. Another important problem is an upper bound of the approximation ratio of RE-PAIR algorithm [7].

References

[1] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the Smallest Grammar: Kolmogorov Complexity in Natural Models. In *Proc. 29th Ann. Sympo. on Theory of Computing*, 792-801, 2002.

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

[3] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage System: a Unifying Framework for Compressed Pattern Matching. *Theoret. Comput. Sci.* (to appear).

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

[5] J. C. Kieffer, E.-H. Yang, G. Nelson, and P. Cosman. Universal Lossless Compression via Multilevel Pattern Matching. *IEEE Trans. Inform. Theory*, IT-46(4), 1227–1245, 2000.

[6] D. Knuth. Seminumerical Algorithms. Addison-Wesley, 441-462, 1981.

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

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

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

[10] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proc. 38th Ann. Sympo. on Foundations of Computer Science*, 137-143, 1997.

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

[12] W. Rytter. Application of Lempel-Ziv Factorization to the Approximation of Grammar-Based Compression. In *Proc. 13th Ann. Sympo. Combinatorial Pattern Matching*, 20-31, 2002.

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

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

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

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

Appendix A

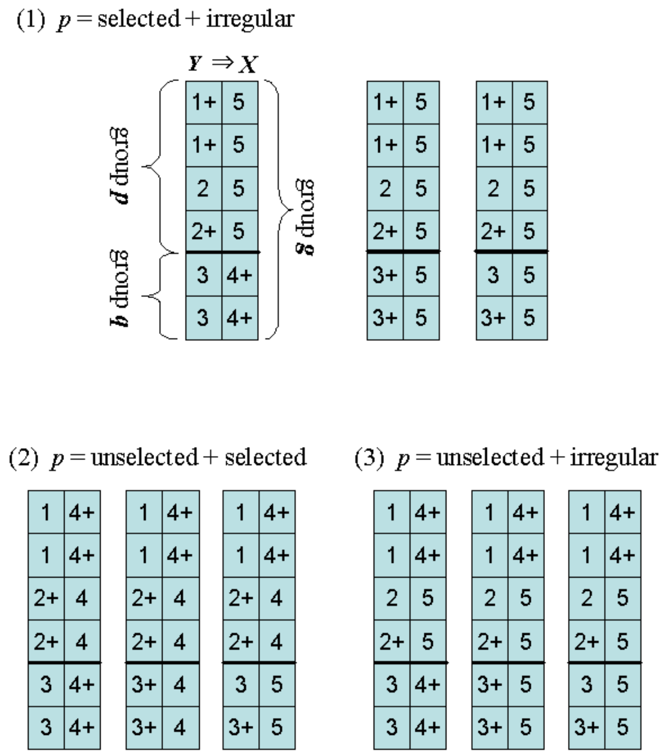

Figure 4: The assignment manner for current segments X. This figure illustrates how the left-fixed segments X are assigned from its left segments Y. The left segments Y is already assigned and then classified into some groups, in this case p and q. The group g for X is obtained from group p and q, which are containing several subgroups. The indices of group p, q, and g are $id = \{1,2\}$, $\{3\}$, $\{4,5\}$. The mark '+' denotes that the marked segments are added to D. For example, on the first case of (1), there is an *unselected* subgroup in Y, then the corresponding segments in X is added to D. Next, there is an *irregular* subgroup and an *unselected* subgroup, then the corresponding segments in X are assigned 4 and 5, respectively. Finally, the remained *selected* subgroup is in the same group of the *irregular* subgroup, then its corresponding segments are assigned 5. In this figure, only the case of q consisting a single subgroup is shown, but this is sufficiently general since the assignment for X is invariable even if q contains other subgroups.

Appendix B

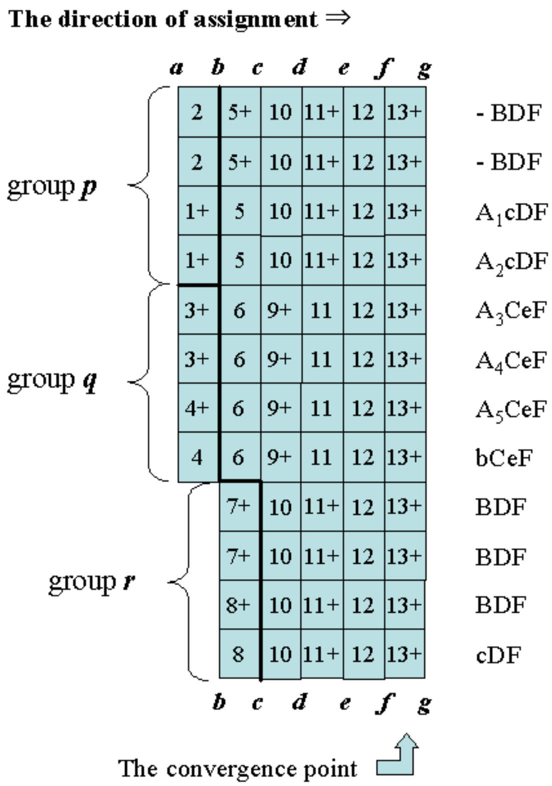

Figure 5: The convergence of assignment for a long *decreasing* prefix case. We assume that a string w contains 8 independent intervals which have the same prefix '*abcdefg*', where this prefix is decreasing. The 1-8 rows represent such 8 intervals. Assume that the set of segments of ab are already classified into two group p and q. The last 4 rows denote other intervals in w which have the same prefix '*bcdefg*'. All 12 rows are merged on the column of cd in a same group. All segments of this group converge to a same *selected* subgroup on the indicated column within the pairs de, ef, and fg are chosen. We note that the convergence of 1-8 rows are guaranteed regardless of the last 4 rows since for each group g', the assignment for right segments of g' is not affected by other groups as long as g' contains 2 subgroups. Finally each interval is compressed in the string shown in its right side. Nonterminals B, C, D, E, F correspond to the production rules $B \to bc, C \to cd, D \to de, E \to ef, F \to fg$, respectively. The '$-$' and '$A_i$' are indefinite since they depend on their left sides.