Simple Linear-Time Off-Line Text Compression by Longest-First Substitution

Ryosuke Nakamura[1], Hideo Bannai[1], Shunsuke Inenaga[2], and Masayuki Takeda[1,3]

[1] Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
[2] Department of Computer Science and Communication Engineering, Kyushu University, Fukuoka 819-0395, Japan
[3] SORST, Japan Science and Technology Agency (JST)
{r-naka, bannai, shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp

Abstract

We consider grammar based text compression with longest first substitution, where non-overlapping occurrences of a longest repeating substring of the input text are replaced by a new non-terminal symbol. We present a new text compression algorithm by simplifying the algorithm presented in [4]. We give a new formulation of the correctness proof introducing the sparse lazy suffix tree data structure. We also present another type of longest first substitution strategy that allows better compression. We show results of preliminary experiments comparing grammar sizes of the two versions of the longest first strategy and the most frequent strategy.

1 Introduction

In this paper we consider text compression by *longest first substitution* (named *LFS*). Given a string, we construct a context-free grammar by substituting new characters for the longest factors that are repeating without overlapping. For example, for $w = \mathtt{abaaabbababb\$}$, we construct the following grammar: $S \rightarrow B\mathtt{aa}ABA\$$; $A \rightarrow \mathtt{abb}$; $B \rightarrow \mathtt{ab}$, which generates only w. Bentley and McIlroy [3] gave an algorithm for this compression scheme, but Nevill-Manning and Witten [7] stated that it does not run in linear time. They also claimed the algorithm by Bentley and McIlroy can be improved so as to run in linear time, but they only noted a too short sketch for how, which is unlikely to give a shape to the idea of the whole algorithm.

In [4], details of an algorithm achieving linear time longest-first substitution were given. The algorithm made extensive use of the suffix tree data structure [9]. However, the algorithm was too complicated for practical implementations, and moreover, the correctness of the algorithm was not easy to see. In this paper, we show a simplified algorithm for longest-first substitution which runs in linear time. The new algorithm is based on the previous algorithm, and uses some properties of suffix trees shown in the previous paper. This paper presents several new tricks and a new, simplified formulation of the algorithm by introducing the sparse lazy suffix tree data structure.

Moreover, this paper deals with another type of longest-first text compression (named *LFS2*), where we also consider repeating factors of the righthand of the existing production rules. This method allows better compression since the total grammar size becomes smaller. For the running string abaaabbababb\$, we obtain the following grammar: $S \to B\mathtt{aa}ABA\$$; $A \to B\mathtt{b}$; $B \to \mathtt{ab}$. This paper presents the first algorithm that accomplishes LFS2 in linear time and space.

1.1 Related Work

It is well known that, given a string w, computing the smallest grammar that generates w is NP-hard [8]. Thus, linear-time greedy strategies for text compression is of significant practical importance. Larsson and Moffat [6] presented a linear-time algorithm called *Re-pair*. It recursively substitutes non-terminal symbols for the factors of length 2 whose non-overlapping occurrences are the most among the factors of length 2. This strategy is called the most frequent first substitution (called *MFFS*). Apostolico and Lonardi [1] presented a text compression technique where factors of the largest "area" are recursively replaced by non-terminal symbols. Here the area of a factor refers to the product of the length of the factor by the number of its non-overlapping occurrences in the input string. To the best of our knowledge, no linear-time algorithm for largest-area-first-substitution based compression is known.

Figure 1 shows a table that compares the grammar sizes generated by MFFS, LFS, and LFS2. The input texts files are from the Canterbury Corpus [2].

	Size	total grammar size		
File	(Bytes)	MFFS	LFS	LFS2
alice29.txt	152090	**38750**	88333	45225
asyoulik.txt	125179	**35245**	74747	41755
cp.html	24603	8006	14559	**7977**
fields.c	11150	3535	6525	**3307**
grammar.lsp	3721	1597	2332	**1431**
kennedy.xls	1029744	**165589**	291536	166250
lcet10.txt	426754	**84923**	235112	103602
plrabn12.txt	481861	**116128**	276714	144078
ptt5	513216	**42813**	266040	47885
sum	38240	13023	20846	**12103**
xargs.1	4227	2096	2772	**1906**

Figure 1. Comparison of the grammar sizes generated by the greedy text compression algorithms.

2 Preliminaries

2.1 Notations

Let Σ be a finite *alphabet* of symbols. We assume that Σ is fixed and $|\Sigma|$ is constant. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, respectively.

The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Also, we assume that all strings end with a unique symbol \$ $\in \Sigma$ that does not occur anywhere else in the strings. Let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the factor of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i : j] = \varepsilon$ for $j < i$, and $w[i :] = w[i : |w|]$ for $1 \leq i \leq |w|$. For any strings x, w, let $BP_w(x)$ denote the set of the beginning positions of all the occurrences of x in w. That is, $BP_w(x) = \{i \mid x = w[i : i + |x| - 1]\}$.

We say that strings x, y *overlap* in w if there exist integers i, j such that $x = w[i : i + |x| - 1]$, $y = w[j : j + |y| - 1]$, and $i \leq j \leq i + |x| - 1$.

Let $\#occ_w(x)$ denote the possible maximum number of *non-overlapping* occurrences of x in w. If $\#occ_w(x) \geq 2$, then x is said to be *repeating* in w. We abbreviate a *longest* repeating factor of w to an *LRF* of w. Remark that there can exist more than one LRF for w.

Let Σ and Π be the set of terminal and non-terminal symbols, respectively, so that $\Sigma \cap \Pi = \emptyset$. A context free grammar $\mathcal{G}$ is a formal grammar in which every production rule is of the form $A \to v$, where $A \in \Pi$ and $v \in (\Sigma \cup \Pi)^*$. The *length* of production rule $A \to v$, denoted $|A| = |v|$, is the length of the string $x \in \Sigma^*$ derived from the production rule. The *size* of the production rule is the number of terminal and non-terminal symbols v contains. Any non-terminal symbol A in $\mathcal{G}$ has $|A|$ positions. We denote $A[1] = A$ and $A[i] = \bullet$ for any $1 < i \leq |A|$. Similarly, any string $w \in (\Sigma \cup \Pi)^*$ has $|w|$ positions and $w[i] \in \Sigma \cup \Pi \cup \{\bullet\}$ denotes the symbol of the i-th position of w.

2.2 Data Structures

Our text compression algorithm uses a data structure based on suffix trees [9]. The suffix tree of string w, denoted by $STree(w)$, is defined as follows:

Definition 1 (Suffix Trees) *$STree(w)$ is a tree structure such that: (1) every edge is labeled by a non-empty factor of w, (2) every internal node has at least two child nodes, (3) all out-going edge labels of every node begin with mutually distinct symbols, and (4) every suffix of w is spelled out in a path starting from the root node.*

Assuming any string w terminates with the unique symbol `$` not appearing elsewhere in w, there is a one-to-one correspondence between a suffix of w and a leaf node of $STree(w)$. It is easy to see that the numbers of the nodes and edges of $STree(w)$ are linear in $|w|$. Moreover, by encoding every edge label x of $STree(w)$ with an ordered pair (i, j) of integers such that $x = w[i : j]$, each edge only needs constant space. Therefore, $STree(w)$ can be implemented with total of $O(|w|)$ space. Also, it is well known that $STree(w)$ can be constructed in $O(|w|)$ time (e.g. see [9]).

$STree(w)$ for string $w =$ `ababa$` is shown in Figure 2. For any node v of $STree(w)$, $str(v)$ denotes the string obtained by concatenating the labels of the edges in the path from the root node to node v. The *length* of node v, denoted $len(v)$, is defined to be $|str(v)|$. It is an easy application of the Ukkonen algorithm [9] to compute the lengths of all nodes while constructing suffix trees. The leaf node ℓ such that $str(\ell) = w[i :]$ is denoted by $leaf_i$, and i is said to be the *id* of the leaf. Every node v of $STree(w)$ except for the root node has a *suffix link*, denoted by $suf(v)$, such that $suf(v) = v'$ where $str(v')$ is a suffix of $str(v)$ and $len(v') + 1 = len(v)$. Linear-time suffix tree construction algorithms (e.g., [9]) make extensive use of the suffix links.

A *sparse suffix tree* [5] of $w \in \Sigma^*$ is a kind of suffix tree which represents only a subset of the suffixes of w. The sparse suffix tree of $w \in (\Sigma \cup \Pi)^*$ represents a subset $\{w[i :] \mid w[i] \in \Sigma\}$. Let ℓ be the length of the LRFs of w. A *reference node* v of the sparse suffix tree of $w \in (\Sigma \cup \Pi)^*$ is a node such that $len(v) \geq \ell + 1$, and there is no node u such that $str(u)$ is a proper prefix of $str(v)$ and $len(u) \geq \ell + 1$.

Our algorithm uses the following data structure.

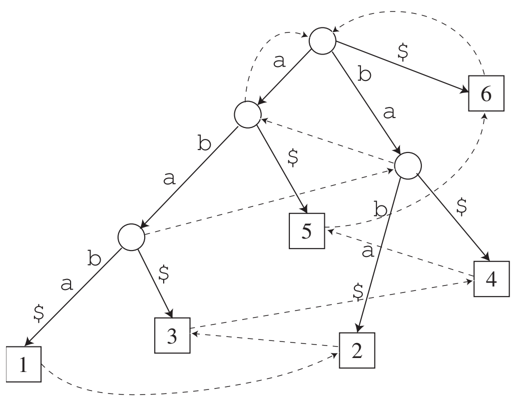

Figure 2. $STree(w)$ **with** $w = \texttt{ababa\$}$**. Solid arrows represent edges, and dotted arrows are suffix links.**

Definition 2 (Sparse Lazy Suffix Trees) *A* sparse lazy suffix tree *(*SLSTree*) of string* $w \in (\Sigma \cup \Pi)^*$*, denoted by* $SLSTree(w)$*, is a kind of sparse suffix tree such that: (1) All paths from the root node to the reference nodes coincide with those of the sparse suffix tree of* w*, and (2) Every reference node* v *stores an ordered triple* $\langle \min(v), \max(v), \mathrm{card}(v) \rangle$ *such that* $\min(v) = \min BP_w(str(v))$*,* $\max(v) = \max BP_w(str(v))$*, and* $\mathrm{card}(v) = \big|BP_w(str(v))\big|$*.*

Proposition 1 *For any string* $w \in \Sigma^*$*,* $SLSTree(w)$ *can be obtained from* $STree(w)$ *in* $O(|w|)$ *time.*

Proof. By a standard postorder traversal on $STree(w)$, propagating the id of each leaf node. □

Since $STree(w)$ can be constructed in $O(|w|)$ time [9], we can build $SLSTree(w)$ in total of $O(|w|)$ time.

3 Off-Line Compression by Longest-First Substitution

Given a text string $w \in \Sigma^*$, we here consider a greedy approach to construct a context-free grammar which generates only w. The key is how to select a factor of w to be replaced by a non-terminal symbol from Π. Here, we consider the *longest-first-substitution* approach where we recursively replace as many LRFs as possible with non-terminal symbols.

Example. Let $w = \texttt{abaaabbababb\$}$. At the beginning, the grammar is of the following simple form $S \to \texttt{abaaabbababb\$}$, where the righthand of the production rule consists only of terminal symbols from Σ. Now we focus on the righthand of S which has two LRFs aba and abb. Let us here choose abb for being replaced by non-terminal $A \in \Pi$, and then we obtain the following grammar: $S \to \texttt{abaa}A\texttt{ab}A\$$; $A \to \texttt{abb}$. The other LRF aba of length 3 is no longer present in the righthand of S. Thus we focus on an LRF ab of length 2. Replacing ab by non-terminal $B \in \Pi$ results in the following grammar: $S \to B\texttt{aa}ABA\$$; $A \to \texttt{abb}$; $B \to \texttt{ab}$. Since the righthand of S has no repeating factor longer than 1, we are done.

Let $w_0 = w$, and let w_k denote the string obtained by replacing an LRF of w_{k-1} with a non-terminal symbol A_k. $LRF(w_{k-1})$ denotes the LRF of w_{k-1} that is replaced by A_k, namely, we create a new production rule $A_k \to LRF(w_{k-1})$. In the above example, $w_0 = w = \texttt{abaaabbababb\$}$, $LRF(w_0) = \texttt{abb}$, $A_1 = A$, $w_1 = \texttt{abaa}A\texttt{ab}A\$$, $LRF(w_1) = \texttt{ab}$, $A_2 = B$, and $w_2 = B\texttt{aa}ABA\$$.

Due to the property of the longest first approach, we have the following observation.

Observation 1 *Let $A_1, \dots, A_k \in \Pi$ be the non-terminal symbols which replace $LRF(w_0)$, $\dots$, $LRF(w_{k-1})$, respectively. For any $1 \leq i \leq k$, the righthand of the production rule of A_i contains none of $A_1, \dots, A_{i-1}$.*

3.1 Algorithm

In this section we show our algorithm which outputs a context free grammar which generates a given string. Our algorithm heavily uses the SLSTree structure.

3.1.1 Finding LRF Using SLSTrees

Lemma 1 *Suppose x is an LRF of w_k represented by none of the nodes of $SLSTree(w_k)$. Then, there exists another LRF y of w_k that is represented by a node of $SLSTree(w_k)$ such that $|x| = |y|$ and $\#occ_{w_k}(y) \geq \#occ_{w_k}(x) = 2$. Moreover, x is no longer present in w_{k+1} after the substitution for y.*

The above lemma implies that it suffices to consider the strings corresponding to the nodes of $SLSTree(w_k)$ as candidates for $LRF(w_k)$.

After constructing $SLSTree(w_0) = SLSTree(w)$, we create a bin-sorted list of the internal nodes of $SLSTree(w)$ in the increasing order of their lengths. It can be done in linear time by a standard traversal on $SLSTree(w)$. We remark that a new internal node v may appear in $SLSTree(w_k)$ for some $k \geq 1$. However, we have that $len(v) \leq LRF(w_{k-1})$. Thus, we can maintain the bin-sorted list by inserting node v in constant time .

Using $SLSTree(w_k)$, for any node v in the bin-sorted list it is easy to determine whether or not $str(s)$ is repeating. Let $s_1, \dots, s_\ell$ be the children of s. If and only if $\max\{\max(s_i) \mid 1 \leq i \leq \ell\} - \min\{\min(s_j) \mid 1 \leq j \leq \ell\} \geq len(s)$, $str(s)$ is repeating. The following lemma can be shown in a similar way as in [4].

Lemma 2 *For any node s of $SLSTree(w_{k-1})$ such that $|LRF(w_k)| \leq len(s) \leq |LRF(w_{k-1})|$, it takes amortized constant time to check whether or not $str(s)$ is an LRF of w_k.*

In what follows we show our greedy strategy of selecting which occurrences of an LRF we replace with a new non-terminal symbol.

The following lemma can be shown by a similar idea mentioned in [4].

Lemma 3 *For any non-repeating factor x of w_k, $BP_{w_k}(x)$ forms a single arithmetic progression.*

Therefore, for any non-repeating factor x of w_k, $BP_{w_k}(x)$ can be expressed by an ordered triple consisting of minimum element $\min BP_{w_k}(x)$, maximum element $\max BP_{w_k}(x)$, and cardinality $|BP_{w_k}(x)|$.

The following lemma is a direct adaptation from [4].

Lemma 4 *Let s be any node of $SLSTree(w_k)$ that is at most as long as the LRFs of w_k, and let $s_1, \dots, s_\ell$ be the children of s. Then $BP_{w_k}(str(s))$ is a disjoint union of $BP_{w_k}(str(s_1))$, $\dots$, $BP_{w_k}(str(s_\ell))$, each forming a single arithmetic progression.*

Lemma 5 *Let s be the node of $SLSTree(w_k)$ such that $str(s)$ is an LRF of w_k, and s' be any child of s. Then, $BP(str(s'))$ contains at most two positions corresponding to non-overlapping occurrences of $str(s)$ in w_k.*

Proof. Assume for contrary that $BP(\mathit{str}(s'))$ contains three occurrences of $\mathit{str}(s)$, and let them be i_1, i_2, i_3 in the increasing order. Then we have

$$i_3 - (i_1 + \mathit{len}(s) - 1) \geq i_3 - i_2 \geq \mathit{len}(s) \geq 1,$$

which implies that $w[i_1 : i_1 + \mathit{len}(s)]$ and $w[i_3 : i_3 + \mathit{len}(s)]$ are non-overlapping. Moreover, since $\mathit{len}(s') > \mathit{len}(s)$ and from Lemma 4, we have $w[i_1 : i_1 + \mathit{len}(s)] = w[i_3 : i_3 + \mathit{len}(s)]$. However, this contradicts that $\mathit{str}(s)$ is an LRF. □

From Lemma 5, each child s' of node s such that $\mathit{str}(s)$ is an LRF, corresponds to at most two non-overlapping occurrences of $\mathit{str}(s)$. Thus, by checking all children $s_1, \ldots, s_\ell$, we can greedily obtain occurrences of $\mathit{str}(s)$ to be replaced, and it takes amortized constant time for each node s.

Note that we have to select occurrences of $\mathit{str}(s)$ so that no occurrences of $\mathit{str}(s)$ remain in the text string, and at least two occurrences of $\mathit{str}(s)$ are selected. We remark that we can greedily choose at least $\max\{2, \#\mathit{occ}(\mathit{str}(s))/2\}$ occurrences.

3.1.2 Updating $\mathit{SLSTree}(w_k^{i-1})$ to $\mathit{SLSTree}(w_k^i)$

Let L be the set of the greedily selected occurrences of $\mathit{LRF}(w_k)$ in w_k. For any $0 \leq i \leq |L|$, w_k^i denote the string obtained after replacing the first i occurrences of $\mathit{LRF}(w_k)$ with non-terminal symbol A_{k+1}. Namely, $w_k^0 = w_k$ and $w_k^{|L|} = w_{k+1}$.

In this section we show how to update $\mathit{SLSTree}(w_k^{i-1})$ to $\mathit{SLSTree}(w_k^i)$. Let p be the beginning position of the i-th largest occurrence in L. Assume that we have $\mathit{SLSTree}(w_k^{i-1})$, and that we have replaced $w_k^{i-1}[p : p+|\mathit{LRF}(w_k)|-1]$ with non-terminal symbol A_{k+1} such that $|A_{k+1}| = |\mathit{LRF}(w_k)|$. We now have w_k^i, and we have to update $\mathit{SLSTree}(w_k^{i-1})$ to $\mathit{SLSTree}(w_k^i)$.

To obtain $\mathit{SLSTree}(w_k^i)$, the most intuitive way is to remove all the suffixes of w_k^{i-1} from the tree and insert all the suffixes of w_k^i into it. However, since only the nodes not longer than $\mathit{LRF}(w_k)$ are important for $\mathit{SLSTree}(w_k^i)$, only the suffixes $w_k^{i-1}[p-t :]$ such that $1 \leq t \leq |\mathit{LRF}(w_k)|$ and $w_k^{i-1}[r] \in \Sigma$ for any $p - t \leq r < p$, have to be removed and only the suffixes $w_k^i[p-t :]$ have to be inserted into the tree.

Proposition 2 *For any $w_k^i[p-t :]$, there is a node s in $\mathit{SLSTree}(w_k^i)$ such that $\mathit{str}(s) = w_k^i[p-t : p-1]$ and s has an edge labeled with $w_k^i[p :] = A_k w_k^i[p + |A_k| :]$ and leading to leaf_{p-t}. Moreover, this edge never exists in $\mathit{SLSTree}(w_k^{i-1})$.*

Lemma 6 *For each t, we can locate node s such that $\mathit{str}(s) = w_k^i[p-t : p-1]$ in amortized constant time.*

Let v be the reference node in the path from the root to some leaf_{p-t}. Assume that leaf_{p-t} is removed from the subtree of v, and redirected to node s in the same path, such that $\mathit{str}(s) = w_k^i[p-t : p-1]$. In order to update $\mathit{SLSTree}(w_k^{i-1})$ to $\mathit{SLSTree}(w_k^i)$, we have to maintain triple $\langle \min(v), \max(v), \mathrm{card}(v) \rangle$ for node v. One may be concerned that if $p - t$ is neither $\min(v)$ or $\max(v)$ and $\mathrm{card}(v) \geq 4$ in $\mathit{SLSTree}(w_k^{i-1})$, the occurrences of $\mathit{str}(v)$ in $\mathit{SLSTree}(w_k^i)$ do not form a single arithmetic progression any more. However, we have the following lemma. For any factor y of w_k^i, let $\mathit{Dead}_{w_k^i}(y) = BP_{w_k^{i-1}}(y) \backslash BP_{w_k^i}(y)$, namely, $\mathit{Dead}_{w_k^i}(y)$ denotes the occurrences of y in w_k^{i-1} that overlap with the i-th greedily selected occurrence of $\mathit{LRF}(w_k)$ in w_k.

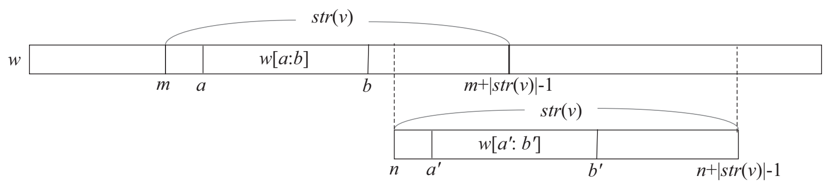

Figure 3. Illustration of proof for Lemma 7.

Lemma 7 *Let v be any reference node of $\mathit{SLSTree}(w_k^i)$ such that $\#occ_{w_k^i}(str(v)) = 1$. For any integer m, n, if $m, n \in BP_{w_k^i}(str(v))$, then there is no integer r such that $m < r < n$ and $r \in Dead_{w_k^i}(str(v))$. (See Figure 3.)*

Proof. Assume for contrary that there exists integer r such that $r \in Dead_{w_k^i}(str(v))$ and $m < r < n$. Since $r \in Dead_{w_k^i}(str(v))$, there exist integers a, b such that $a \le r \le b$, and $b - a + 1 = 2|LRF(w_k)|$. For any integer j such that $a \le j \le b$ and $j \in BP_{w_k^{i-1}}(str(v))$, we have $j \in Dead_{w_k^i}(str(v))$. Since $m, n \notin Dead_{w_k^i}(str(v))$, $m < a < b < n$. As $str(v)$ is non-repeating, $n < m + len(v) - 1$. Since $m < a < b < m + len(v) - 1$, $w[a : b]$ is a factor of $str(v)$. Therefore, there exist two integers a', b' such that $w[a' : b'] = w[a : b]$. Since $m < a < b < n < a' < b' < n + len(v) - 1$, $w[a : b]$ is repeating and $|w[a : b]| = b - a + 1 = 2|LRF(w_k)| > |LRF(w_k)|$. It contradicts that $LRF(w_k)$ is an LRF of w_k. $\square$

Recall that p is the beginning position of the i-th largest greedily selected occurrence of $LRF(w_k)$ in w_k. Also, for any $1 \le t \le |LRF(w_k)|$ such that $w_k^{i-1}[r] \in \Sigma$ for every $p - t \le r < p$, we have removed leaf_{p-t} from the subtree rooted at the reference node v and have reconnected it to node s such that $str(s) = w_k^i[p - t : p - 1]$. According to the above lemma, if $\min(v) < p - t < \max(v)$, leaf_j for every $p - t \le j \le \max(v)$ is removed from the subtree of v. After processing leaf_{p-t}, then $\max(v)$ is updated to $p - t - d$ where $d = (\min(v) + \max(v))/\mathrm{card}(v)$ is the step of the progression, and $\mathrm{card}(v)$ is updated to $(\max(v) - (p - t))/d + 1$.

Notice that leaf_{p+h} for every $0 \le h \le |LRF(w_k)| - 1$ has to be removed from the tree, since $w_k^i[p + h] \notin \Sigma$ and therefore this leaf node should not exist in $\mathit{SLSTree}(w_k^i)$. Removing each leaf can be done in constant time. Maintaining the information about the triple for the arithmetic progression of the reference nodes can be done in the same way as mentioned above.

The following lemma states how to locate each reference node.

Lemma 8 *Let p be the i-th greedily selected occurrence of $LRF(w_k)$ in w_k. For any integer ℓ such that $w_k^{i-1}[\ell] \in \Sigma$, let $v(\ell)$ denote the reference node of $\mathit{SLSTree}(w_k^{i-1})$ in the path from the root spelling out suffix $w_k^{i-1}[\ell :]$. For each j such that $p - |LRF(w_k)| \le j \le p + |LRF(w_k)| - 1$, we can locate the reference node $v(j)$ in amortized constant time.*

Proof. Let $\ell = |LRF(w_k)|$. We find $v(p - \ell)$ by spelling out $w_k^{i-1}[p - \ell :]$ from the root in $O(\ell)$ time, since there can be at most $\ell + 1$ nodes in the path from the root to $v(p - \ell)$.

Suppose we have found $v(j - 1)$. We find $v(j)$ as follows. Let $u(j - 1)$ be the parent node of $v(j - 1)$. We have $len(u(j - 1)) \le \ell$ and $len(v(j - 1)) \le \ell + 1$. We go to $\mathit{suf}(u(j - 1))$. Since $len(\mathit{suf}(u(j - 1))) + 1 = len(u(j - 1))$, we have $len(\mathit{suf}(u(j - 1))) \le \ell + 1$. Thus, we can find $v(j)$ by going down the path starting from $\mathit{suf}(u(j - 1))$ and spelling out $w_k^{i-1}[j - 1 + len(u(j - 1)) : j - 1 + len(v(j - 1))] = w_k^{i-1}[j + len(\mathit{suf}(u(j - 1))) : j - 1 + len(v(j - 1))]$. (See also the left illustration of Figure 4.)

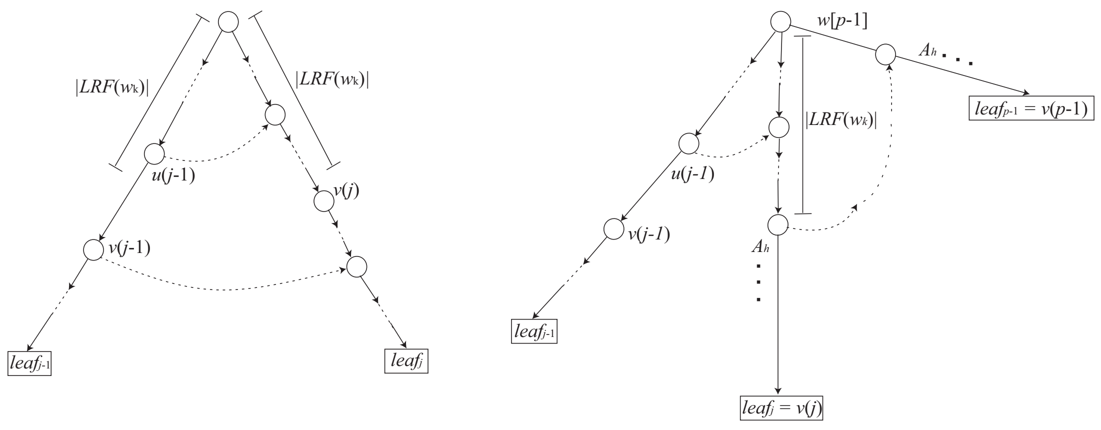

Figure 4. The left figure illustrates how to find $v(j)$ from $v(j-1)$. The right one illustrates a special case where $v(j) = \mathit{leaf}_j$. Once $v(j) = \mathit{leaf}_j$, it stands that $v(k) = \mathit{leaf}_k$ for any $j \leq k \leq p-1$.

A special case happens when there exists a node s in the path from the root to leaf_j, such that $\mathit{len}(s) = \ell$ and the edge from s in the path starts with some non-terminal symbol A_h with $h < k$. Namely, $w_k^i[j+\ell] = A_h$. Due to the property of the longest first approach, we have $|A_h| \geq \ell$. Thus $v_j = \mathit{leaf}_j$. Moreover, for any $j \leq k \leq p-1$, $v(k) = \mathit{leaf}_k$. (See also the right illustration of Figure 4.) It is thus clear that each $v(k)$ can be found in constant time. Since $|A_h| \geq \ell = \mathit{LRF}(w_k)$, the leaves corresponding to $w_k^{i-1}[p+x-1:]$ with $1 \leq x \leq \ell$ do not exist in $\mathit{SLSTree}(w_k^{i-1})$. $\square$

From the above discussions, we conclude that:

Theorem 1 *For any string $w \in \Sigma^*$, the proposed algorithm for text compression by longest first substitution runs in $O(|w|)$ time using $O(|w|)$ space.*

Pseudo-codes of our algorithms are shown in Algorithms 1, 2 and, 3.

3.2 Reducing Grammar Size

In the above sections we considered text compression by longest first substitution, where we construct a context free grammar $\mathcal{G}$ that generates only a given string w. By Observation 1, for any production rule $A_k \to x_k$ of $\mathcal{G}$, x_k contains only terminal symbols from Σ. In this section, we take the factors of x_k into consideration for candidates of LRFs, and also replace LRFs appearing x_k. This way we can reduce the total size of the grammar. In so doing, we consider an LRF of string $z_k = w_k x_1 \$_1 \cdots x_k \$_k$, where $z_0 = w_0 = w$ and each $\$_i$ appears nowhere else in z_k.

Example. Let $w = w_0 = z_0 = \texttt{abaaabbababb}\$_0$. We replace an LRF abb with A, and obtain the following grammar: $S \to \texttt{abaa}A\texttt{ab}A\$_0$; $A \to \texttt{abb}$. Then, $w_1 = \texttt{abaa}A\texttt{ab}A\$_0$ and $\mathit{LRF}(z_0) = \texttt{abb}$. Now, $z_1 = \texttt{abaa}A\texttt{ab}A\$_0\texttt{abb}\$_1$. We replace an LRF ab of z_1 with a non-terminal B, getting $S \to B\texttt{aa}ABA\$_0$; $A \to B\texttt{b}$; $B \to \texttt{ab}$. Then, $w_2 = B\texttt{aa}ABA\$_0$ and $\mathit{LRF}(z_1) = \texttt{ab}$. Now, $z_2 = B\texttt{aa}ABA\$_0B\texttt{b}\$_1\texttt{ab}\$_2$. Since there is no LRF of length more than 1 in z_2, we are done.

We call this new method of text compression *LFS2*.

Theorem 2 *Given a string w, the LFS2 strategy compresses w in linear time and space.*

Algorithm 1: Recursively find longest repeating factors.

```
Input: String w ending with a unique symbol
Output: Set of grammar rules which produce w, greedily selected by substituting longest
        repeating factors
SLSTree := sparse lazy suffix tree of w; bins := bin-sorted nodes; len := |w|; rules := ∅;
while true do
    while (n = bins.getNextOfLength(len)) = null do
        if len ≤ 2 then return rules;
        foreach x ∈ bins(len) do update x.min, x.max, x.card from children;
        len−−;
    update n.min, n.max, n.card from children;
    if n.max − n.min ≥ n.pathlen /* n is repeating factor */ then
        nonTerm := new non-terminal symbol;
        rules := rules ∪ {nonTerm → n.path };
        gso := getGreedilySelectedOccurrences(n);
        updateSLSTree(w, n.pathlen, nonTerm, gso, SLSTree, bins);
```

Algorithm 2: **updateSLSTree**

```
Input: (w, LRFlen, nonTerm, gso, SLSTree, bins)
foreach occpos ∈ gso do
    for pos = max{1, occpos − LRFlen} to min{|w|, occpos + LRFlen − 1} do
        v := find first node on path to leaf pos such that v.pathlen > LRFlen;
        delete leaf pos; maintain v.card, v.min, v.max;
        if pos < occpos && notDead(pos) then
            s := find/create node on path to leaf pos such that s.pathlen = occpos − pos;
            if s was newly created then bins(s.pathlen).addNode(s);
            recreate leaf pos:⟨min, max, card⟩ = ⟨pos, pos, 1⟩; add edge (s, nonTerm, pos);
        if pos > occpos then w[pos] = •; markDead(pos);
    w[occpos] := nonTerm; markDead(occpos);
return
```

Proof. We modify the algorithm proposed in the previous sections. If we have a generalized SLSTree for set $\{w_k,\ x_1\$_1, \ldots, x_k\$_k\}$ of strings, we can find an LRF of $z_k = w_k x_1 \$_1 \cdots x_k \$_k$. It follows from the property of the longest first substitution strategy that $|x_i| \geq |x_j|$ for any $i < j$. Therefore, any new node inserted into the generalized SLSTree for $\{w_k,\ x_1\$_1, \ldots, x_{k-1}\$_{k-1}\}$ is shorter than the reference nodes of the tree. Thus, using the Ukkonen on-line algorithm [9], we can obtain the generalized SLSTree of $\{w_k,\ x_1\$_1, \ldots, x_k\$_k\}$, by inserting the suffixes of each $x_k\$_k$ into the generalized SLSTree of $\{w_k,\ x_1\$_1, \ldots, x_{k-1}\$_{k-1}\}$ in $O(|x_k\$_k|)$ time. It is easy to see that the total length of $x_1\$_1, \ldots, x_k\$_k, \ldots$ is $O(|w|)$. □

Algorithm 3: **getGreedilySelectedOccurrences**

```
Input: LRFnode
Output: Set of greedily selected occurrences of LRFnode.path
gso := $\emptyset$;
foreach c $\in$ LRFnode.children do
    occ := 0;
    if notDead(c.min) then occ := c.min;
    else occ := find first occurrence of LRFnode.path after c.min + endOfDeadArea[c.min];
    if occ $\neq$ 0 && notDead(occ + LRFnode.pathlen $-1$) then
        gso := gso $\cup$ {occ};
        for pos = occ to occ + LRFnode.pathlen $-1$ do
            markEndOfDeadArea(pos, occ + LRFnode.pathlen $-1$);
        occ := occ + LRFnode.pathlen;
        if notDead(occ) && notDead(occ + LRFnode.pathlen $-1$) then gso := gso $\cup$ {occ};

return gso;
```

References

[1] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744, 2000.

[2] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. Data Compression Conference '97 (DCC'97)*, pages 201–210, 1997.

[3] J. Bentley and D. McIlroy. Data compression using long common strings. In *Proc. Data Compression Conference '99 (DCC'99)*, pages 287–295. IEEE Computer Society, 1999.

[4] S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara. Linear-time off-line text compression by longest-first substitution. In *Proc. SPIRE'03*, volume 2857 of *LNCS*, pages 137–152. Springer-Verlag, 2003.

[5] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, volume 1090 of *LNCS*, pages 219–230. Springer-Verlag, 1996.

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

[7] C. G. Nevill-Manning and I. H. Witten. Online and offline heuristics for inferring hierarchies of repetitions in sequences. *Proc. IEEE*, 88(11):1745–1755, 2000.

[8] J. Storer, NP-completeness Results Concerning Data Compression, Technical report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

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