Algorithms **2009**, *2*, 1429-1448; doi:10.3390/a2041429

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Linear-Time Text Compression by Longest-First Substitution

Ryosuke Nakamura [1], Shunsuke Inenaga [2,⋆], Hideo Bannai [1], Takashi Funamoto [1], Masayuki Takeda [1] and Ayumi Shinohara [3]

[1] Department of Informatics, Kyushu University, 744 Motooka, Fukuoka 819-0395, Japan; E-mails: bannai@inf.kyushu-u.ac.jp (H.B.); takeda@inf.kyushu-u.ac.jp (M.T.)

[2] Graduate School of Information Science and Electrical Engineering, Kyushu University, 744 Motooka, Fukuoka 819-0395, Japan

[3] Graduate School of Information Sciences, Tohoku University, Aoba 6-6-05, Aramaki, Sendai 980-8579, Japan; E-mail: ayumi@ecei.tohoku.ac.jp (A.S.)

⋆ Author to whom correspondence should be addressed; E-mail: inenaga@c.csce.kyushu-u.ac.jp.

Received: 30 September 2009 / Accepted: 20 November 2009 / Published: 25 November 2009

Abstract: We consider grammar-based text compression with *longest first substitution* (*LFS*), where non-overlapping occurrences of a longest repeating factor of the input text are replaced by a new non-terminal symbol. We present the first linear-time algorithm for LFS. Our algorithm employs a new data structure called *sparse lazy suffix trees*. We also deal with a more sophisticated version of LFS, called *LFS2*, that allows better compression. The first linear-time algorithm for LFS2 is also presented.

Keywords: grammar-based text compression, suffix trees, linear-time algorithms

1. Introduction

Data compression is a task of reducing data description length. Not only does it enable us to save space for data storage, but also it reduces time for data communication. This paper focuses on text compression where the data to be compressed are texts (strings). Recent research developments show that text compression has a wide range of applications, e.g., pattern matching [1, 2, 3], string similarity computation [4, 5], detecting palindromic/repetitive structures [4, 6], inferring hierarchal structure of natural language texts [7, 8], and analyses of biological sequences [9].

Grammar-based compression [10] is a kind of text compression scheme in which a context-free grammar (CFG) that generates only an input text w is output as a compressed form of w. Since the problem of computing the smallest CFG which generates w is NP-hard [11], many attempts have been made to develop practical algorithms that compute a small CFG which generates w. Examples of grammar-based compression algorithms are LZ78 [12], LZW [13], Sequitur [7], and Bisection [14]. Approximation algorithms for optimal grammar-based compression have also been proposed [15, 16, 17]. The first compression algorithm based on a subclass of context-sensitive grammars was introduced in [18].

Grammar-based compression based on *greedy* substitutions has been extensively studied. Wolff [19] introduced a concept of *most-frequent-first substitution* (*MFFS*) such that a digram (a factor of length 2) which occurs most frequently in the text is recursively replaced by a new non-terminal symbol. He also presented an $O(n^2)$-time algorithm for it, where n is the input text length. A linear-time algorithm for most-frequent-first substitution, called Re-pair, was later proposed by Larsson and Moffat [20]. Apostolico and Lonardi [21] proposed a concept of *largest-area-first substitution* such that a factor of the largest "area" is recursively replaced by a new non-terminal symbol. 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 text. It was reported in [22] that compression by largest-area-first substitution outperforms gzip (based on LZ77 [23]) and bzip2 (based on the Burrows-Wheeler Transform [24]) on DNA sequences. However, to the best of our knowledge, no linear-time algorithm for this compression scheme is known.

This paper focuses on another greedy text compression scheme called *longest-first substitution* (*LFS*), in which a longest repeating factor of an input text is recursively replaced by a new non-terminal symbol. For example, for input text $w = \texttt{abaaabbababb\$}$, the following grammar

$$\begin{aligned} S &\rightarrow B\texttt{aa}ABA\$; \\ A &\rightarrow \texttt{abb}; \\ B &\rightarrow \texttt{ab}, \end{aligned}$$

which generates only w is the output of LFS.

In this paper, we propose the *first linear-time algorithm* for text compression by LFS substitution. A key idea is the use of a new data structure called *sparse lazy suffix trees*. Moreover, this paper deals with a more sophisticated version of longest-first text compression (named *LFS2*), where we also consider repeating factors of the right-hand of the existing production rules. For the same input text $w = \texttt{abaaabbababb\$}$ as above, we obtain the following grammar:

$$\begin{aligned} S &\rightarrow B\texttt{aa}ABA\$; \\ A &\rightarrow B\texttt{b}; \\ B &\rightarrow \texttt{ab}. \end{aligned}$$

This method allows better compression since the total grammar size becomes smaller. In this paper, we present the first linear-time algorithm for text compression based on LFS2. Preliminary versions of our paper appeared in [25] and [26].

Related Work

It is true that several algorithms for LFS or LFS2 were already proposed, however, in fact none of them runs in linear time in the worst case. Bentley and McIlroy [27] proposed an algorithm for LFS, but Nevill-Manning and Witten [8] pointed out that the algorithm does not run in linear time. Nevill-Manning and Witten also claimed that the algorithm 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. Lanctot et al. [28] proposed an algorithm for LFS2 and stated that it runs in linear time, but a careful analysis reveals that it actually takes $O(n^2)$ time in the worst case for some input string of length n. See Appendix for our detailed analysis.

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 $i \leq j \leq i + |x| - 1$ or $j \leq i \leq j + |y| - 1$.

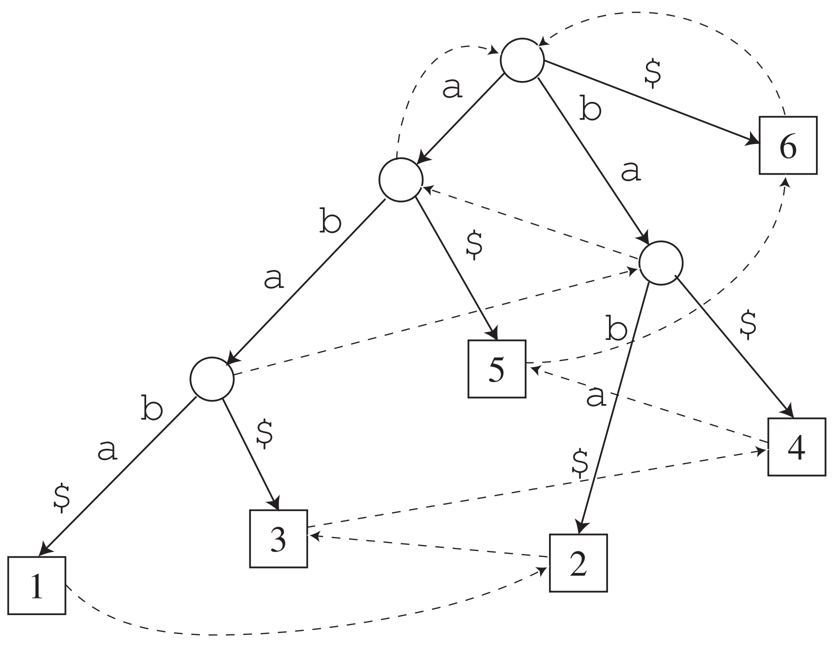
Let $\#occ_w(x)$ denote the possible maximum number of *non-overlapping* occurrences of x in w. If $\#occ_w(x) \geq 2$, then x is said to be *repeating* in w. We abbreviate a *longest* repeating factor of w to an *LRF* of w. Remark that there can exist more than one LRF for w.

Let Σ and Π be the set of terminal and non-terminal symbols, respectively, such that $\Sigma \cap \Pi = \emptyset$. A context-free grammar $\mathcal{G}$ is a formal grammar in which every production rule is of the form $A \to u$, where $A \in \Pi$ and $u \in (\Sigma \cup \Pi)^*$. Let $u = xBy$ and $v = x\beta y$ with $x, y, \beta \in (\Sigma \cup \Pi)^*$ and $B \in \Pi$. If there exists a production rule $B \to \beta$ in $\mathcal{G}$, then $v = x\beta y$ is said to be directly derived from $u = xBy$ by $\mathcal{G}$, and it is denoted by $u \Rightarrow_{\mathcal{G}} v$. If there exists a sequence $w_0, w_1, \ldots, w_n$ such that $w_i \in (\Sigma \cup \Pi)^*$ and

$$u = w_0 \Rightarrow_{\mathcal{G}} w_1 \Rightarrow_{\mathcal{G}} \cdots \Rightarrow_{\mathcal{G}} w_n = v,$$

then we say that v is derived from u. The *length* of a non-terminal symbol A, denoted $|A|$, is the length of the string $z \in \Sigma^*$ that is derived from the production rule $A \to v$. For convenience, we assume that any non-terminal symbol A in $\mathcal{G}$ has $|A|$ positions. The *size* of the production rule is the number of terminal and non-terminal symbols v contains.

Figure 1. $STree(w)$ with $w = \texttt{ababa\$}$. Solid arrows represent edges, and dotted arrows are suffix links.



2.2. *Data Structures*

Our text compression algorithm uses a data structure based on suffix trees [29]. 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 [29]).

$STree(w)$ for string $w = \texttt{ababa\$}$ is shown in Figure 1. 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 [29] to compute the lengths of all nodes while constructing $STree(w)$. 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., [29]) make extensive use of the suffix links.

A *sparse suffix tree* [30] 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 the subset $\{w[i:] \mid w[i] \in \Sigma\}$ of suffixes of w which begin with a terminal symbol. Let ℓ be the length of the LRFs of w. A *reference node* of the sparse suffix tree of $w \in (\Sigma \cup \Pi)^*$ is any node v 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.

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* $\mathrm{card}(v) = \left|BP_w(str(v))\right|$.

$SLSTree(w)$ is called "lazy" since its subtrees that are located below the reference nodes may not coincide with those of the corresponding sparse suffix tree of w. Our algorithms of Section 3. run in linear time by "neglecting" updating these subtrees below the reference nodes.

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 [29], 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 $S \rightarrow \texttt{abaaabbababb\$}$, where the right-hand of the production rule consists only of terminal symbols from Σ. Now we focus on the right-hand of S which has two LRFs `aba` and `abb`. Let us here choose `abb` to be replaced by non-terminal $A \in \Pi$. We obtain the following grammar: $S \rightarrow \texttt{abaa}A\texttt{ab}A\$$; $A \rightarrow \texttt{abb}$. The other LRF `aba` of length 3 is no longer present in the right-hand 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: $S \rightarrow B\texttt{aa}ABA\$$; $A \rightarrow \texttt{abb}$; $B \rightarrow \texttt{ab}$. Since the right-hand 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 $A_k \rightarrow LRF(w_{k-1})$. 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 $A_1, \ldots, A_k \in \Pi$ be the non-terminal symbols which replace $LRF(w_0), \ldots, LRF(w_{k-1})$, respectively. For any $1 \le i \le k$, the right-hand of the production rule of A_i contains none of $A_1, \ldots, A_{i-1}$.*

In what follows, we will show our algorithm which outputs a context-free grammar which generates a given string. Our algorithm heavily uses the SLSTree structure.

3.1. How to Find $LRF(w_k)$ Using $SLSTree(w_k)$

In this section, we show how to find an LRF of w_k from $SLSTree(w_k)$.

The next lemmas characterize an LRF of w_k that is *not* represented by a node of $SLSTree(w_k)$.

Lemma 1 *If an LRF x of w_k is not represented by a node of $SLSTree(w_k)$, then $\max BP_{w_k}(x) = \min BP_{w_k}(x) + |x|$.*

Proof. Let $i = \min BP_{w_k}(x)$ and $j = \max BP_{w_k}(x)$. Since x is a repeating factor of w_k, $|BP_{w_k}(x)| \ge 2$, which means that $i \ne j$. If $w_k[i + |x|] \ne w_k[j + |x|]$, then it contradicts the precondition that x is not represented by a node of $SLSTree(w_k)$. Hence we have $w_k[i + |x|] = w_k[j + |x|]$. Moreover, since x is an LRF of w_k, we have $j \ge i + |x|$. However, if we assume $j > i + |x|$, this contradicts the precondition that x is an LRF of w_k, since $w_k[i + |x|] = w_k[j + |x|]$ and we obtain a longer LRF $w_k[i : i + |x|] = w_k[j : j + |x|]$. Hence we have $j = i + |x|$. □

The above lemma implies that an LRF x is not represented by a node of $SLSTree(w_k)$ only if the first and the last occurrences of x form a *square* xx in w_k. For example, see Figure 1 that illustrates $SLSTree(w_0)$ for $w = \texttt{ababa\$}$. One can see that ab is an LRF of w_0 but it is not represented by a node of $SLSTree(w_0)$.

However, the following lemma guarantees that it is indeed sufficient to consider the strings represented by nodes of $SLSTree(w_k)$ as candidates for $LRF(w_k)$.

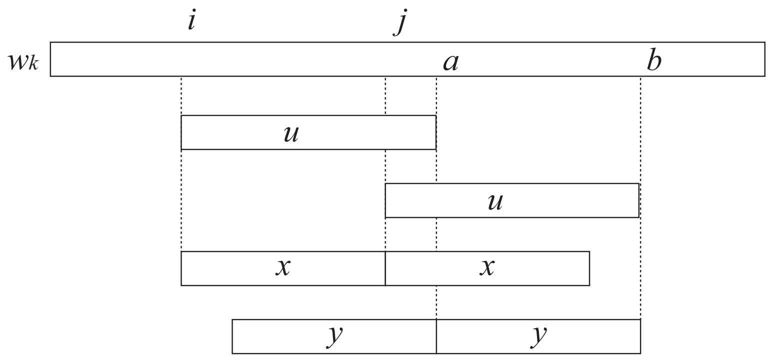
Lemma 2 *Let x be an LRF of w_k that is not represented by a node of $SLSTree(w_k)$. Then, there exists another LRF y of w_k that is represented by a node of $SLSTree(w_k)$ such that $|x| = |y|$. Moreover, x is no longer present in w_{k+1} after a substitution for y (see also Figure 2).*

Proof. Let $i = \min BP_{w_k}(x)$ and $j = \max BP_{w_k}(x)$. It follows from Lemma 1 that $j = i + |x|$. Suppose that x is represented on an edge from some node s to some node t of $STree(w)$. Let $u = str(t)$. Then we have $BP_{w_k}(x) = BP_{w_k}(u)$. Let y be the suffix of u of length $|x|$. It is clear that $i + |u| - |y|, j + |u| - |y| \in BP_{w_k}(y)$. Since $j = i + |x| = i + |y|$, $\#occ_{w_k}(y) \ge 2$. Thus y is an LRF of w_k. Since u is represented by node t and $i = \min BP_{w_k}(u)$ and $j = \max BP_{w_k}(u)$, we know that $w_k[i + |u|] \ne w_k[j + |u|]$. Hence y is represented by a node of $SLSTree(w_k)$. Since x occurs only within the region $w_k[i : j + |u| - 1]$, x does not occur in w_{k+1} after a substitution for y. □

In the running example of Figure 1, ba is an LRF of w_0 that is represented by a node of $SLSTree(w_0)$. After its two occurrences are replaced by a non-terminal symbol A_1, then ab, which is an LRF of w_0 not represented by a node of $SLSTree(w_0)$, is no more present in $w_1 = \texttt{a}A_1A_1\$$.

After constructing $SLSTree(w_0) = SLSTree(w)$, we create a bin-sorted list of the internal nodes of $SLSTree(w)$ in the decreasing order of their lengths. This can be done in linear time by a standard

Figure 2. Illustration for proof of Lemma 2. Since u is represented by a node of $SLSTree(w_k)$, we know that $w_k[i + |u|] \neq w_k[j + |u|]$.



traversal on $SLSTree(w)$. We remark that a new internal node v may appear in $SLSTree(w_k)$ for some $k \geq 1$, which did not exist in $SLSTree(w_{k-1})$. 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.

Given a node s in the bin-sorted list, we can determine whether $str(s)$ is repeating or not by using $SLSTree(w_k)$, as follows.

Lemma 3 *Let s be any node of $SLSTree(w_k)$ with $len(s) \leq |LRF(w_k)|$ and let $s_1, \ldots, s_\ell$ be the children of s. Then $BP_{w_k}(str(s))$ is a disjoint union of $BP_{w_k}(str(s_1)), \ldots, BP_{w_k}(str(s_\ell))$.*

Proof. Clear from the definition of $SLSTree(w_k)$. □

Lemma 4 *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.*

Proof. Let $s_1, \ldots, s_\ell$ be the children of s. Then, $str(s)$ is repeating if and only if

$$\max\{\max BP_{w_{k-1}}(s_i) \mid 1 \leq i \leq \ell\} - \min\{\min BP_{w_{k-1}}(s_j) \mid 1 \leq j \leq \ell\} \geq len(s).$$

Remark that the values of $\min BP_{w_{k-1}}(s_i)$ and $\max BP_{w_{k-1}}(s_i)$ are stored in node s_i and can be referred to in constant time. Since the above inequality is checked at most once for each node s, it takes amortized constant time. □

Suppose we have found an LRF of w_k as mentioned above. In the sequel, we show our greedy strategy to select occurrences of the LRF in w_k to be replaced with a new non-terminal symbol.

The next lemma is essentially the same as Lemma 2 of Kida et al. [1].

Lemma 5 *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 $|BP_{w_k}(x)|$, which takes constant space.

Lemma 6 *Let s be any node of $SLSTree(w_k)$ such that $str(s)$ is an LRF of w_k, and s' be any child of s. Then, $BP_{w_k}(str(s'))$ contains at most two positions corresponding to non-overlapping occurrences of $str(s)$ in w_k.*

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

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

which implies that $w_k[i_1 : i_1 + len(s)]$ and $w_k[i_3 : i_3 + len(s)]$ are non-overlapping. Moreover, since $len(s') > len(s)$, we have $w_k[i_1 : i_1 + len(s)] = w_k[i_3 : i_3 + len(s)]$. However, this contradicts the precondition that $str(s)$ is an LRF of w_k. □

From Lemma 6, each child s' of node s such that $str(s)$ is an LRF, corresponds to at most two non-overlapping occurrences of $str(s)$. Due to Lemma 3, we can greedily select occurrences of $str(s)$ to be replaced by a new non-terminal symbol, by checking all children $s_1, \ldots, s_\ell$ of node s. According to Lemma 5, it takes amortized constant time to select such occurrences for each node s.

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

3.2. How to Update $SLSTree(w_k^{i-1})$ to $SLSTree(w_k^i)$

Let L be the set of the greedily selected occurrences of $LRF(w_k)$ in w_k. For any $0 \leq i \leq |L|$, let w_k^i denote the string obtained after replacing the first i occurrences of $LRF(w_k)$ 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 $SLSTree(w_k^{i-1})$ to $SLSTree(w_k^i)$. Let p be the beginning position of the i-th occurrence in L. Assume that we have $SLSTree(w_k^{i-1})$, and that we have replaced $w_k^{i-1}[p : p + |LRF(w_k)| - 1]$ with non-terminal symbol A_{k+1} such that $|A_{k+1}| = |LRF(w_k)|$. We now have w_k^i, and we have to update $SLSTree(w_k^{i-1})$ to $SLSTree(w_k^i)$.

A naive way to obtain $SLSTree(w_k^i)$ is to remove all the suffixes of w_k^{i-1} from $SLSTree(w_k^{i-1})$ and insert all the suffixes of w_k^i into it. However, since only the nodes not longer than $LRF(w_k)$ are important for our longest-first strategy, only the suffixes $w_k^{i-1}[p-t :]$ such that $1 \leq t \leq |LRF(w_k)|$ and $w_k^{i-1}[r] \in \Sigma$ for any $p - t \leq r < p$ have to be removed from $SLSTree(w_k^{i-1})$, and only the suffixes $w_k^i[p - t :]$ have to be inserted into the tree (see the light-shaded suffixes of Figure 3).

Lemma 7 *For any t, let r be the shortest node of $SLSTree(w_k^{i-1})$ such that $w_k^i[p - t : p - 1]$ is a prefix of $str(r)$. Assume $p - t = \min BP_{w_k^{i-1}}(str(r))$.*

1. *If $len(r) > |LRF(w_k)| + t - 1$, then there exists an edge in $SLSTree(w_k^i)$ from the root node to $leaf_{p-t}$ labeled with $w_k^i[p - t :]$.*

2. *If $len(r) \leq |LRF(w_k)| + t - 1$, then there exists a node s in $SLSTree(w_k^i)$ such that $str(s) = w_k^i[p - t : p - 1]$ and s has an edge labeled with $w_k^i[p :] = A_k w_k^i[p + |A_k| :]$ and leading to $leaf_{p-t}$.*

Proof. Consider Case 1 (see also Figure 4). Since $t \geq 1$, $len(r) > |LRF(w_k)|$. Hence $str(r)$ is a non-repeating factor of w_k^i. By Lemma 5, $BP_{w_k^{i-1}}(str(r))$ forms a single arithmetic progression. Also, since $len(r) > |LRF(w_k)|$, $\max BP_{w_k^{i-1}}(str(r)) - \min BP_{w_k^{i-1}}(str(r)) \leq |LRF(w_k)|$. Therefore, if

Figure 3. $LRF(w_k)$ at position p of w_k^{i-1} is replaced by non-terminal symbol A_k in w_k^i. Every $w_k^{i-1}[p-t:]$ is removed from the tree and every $w_k^i[p-t:]$ is inserted into the tree (the light-shaded suffixes in the right figure). In addition, every $w_k^{i-1}[p+h:]$ for $1 \le h \le |LRF(w_k)|-1$ is removed from the tree (the dark-shaded suffixes in the right figure).

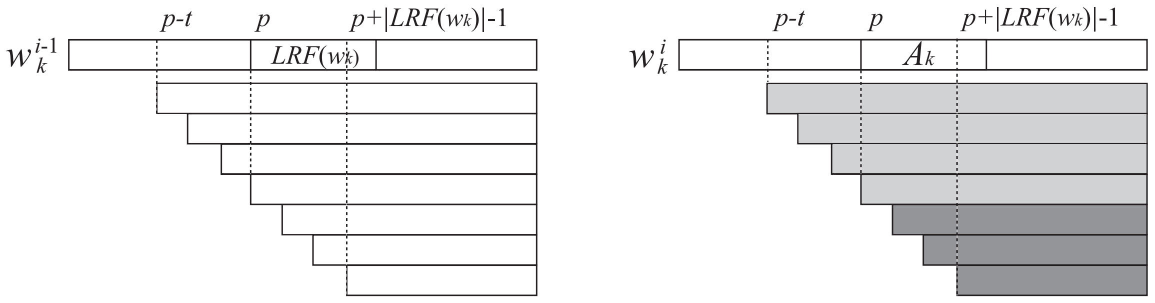
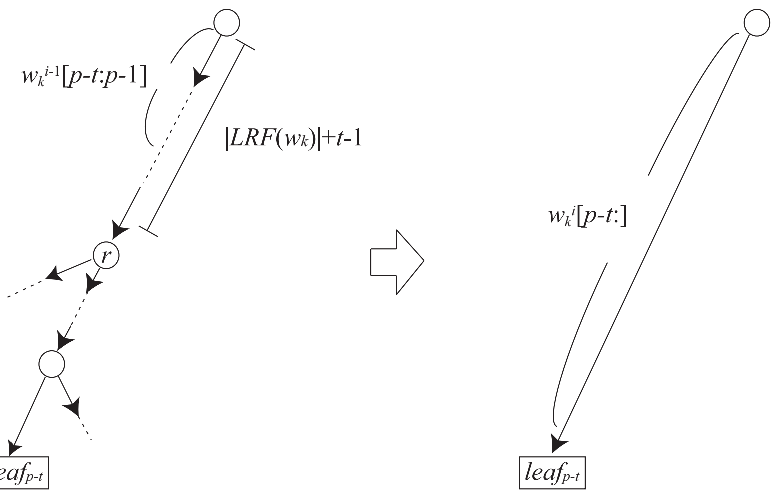


Figure 4. Illustration of Case 1 of Lemma 7.



$p-t = \min BP_{w_k^{i-1}}(str(r))$, then $BP_{w_k^i}(w_k^i[p-t:]) = \{p-t\}$. Hence there exists an edge from the root node to $leaf_{p-t}$ labeled with $w_k^i[p-t:]$ in $SLSTree(w_k^i)$.

Consider Case 2 (see also Figure 5). Let $u = w_k^{i-1}[p-t:p-1] = w_k^i[p-t:p-1]$. Then $|u| = t-1$. Since $len(r) \le |LRF(w_k)|+t-1$, and since r is not longer than the reference node in the path spelling out $uLRF(w_k)$ from the root node of $SLSTree(w_k^i)$, there exists at least one integer m such that $m \in BP_{w_k^i}(str(r))$ and $m \notin BP_{w_k^i}(uA_k)$. Hence there exists a node s in $SLSTree(w_k^i)$ such that $str(s) = u$ and has an out-going edge labeled with $w_k^i[p:] = A_k w_k^i[p+|A_k|:]$ and leading to $leaf_{p-t}$.
□

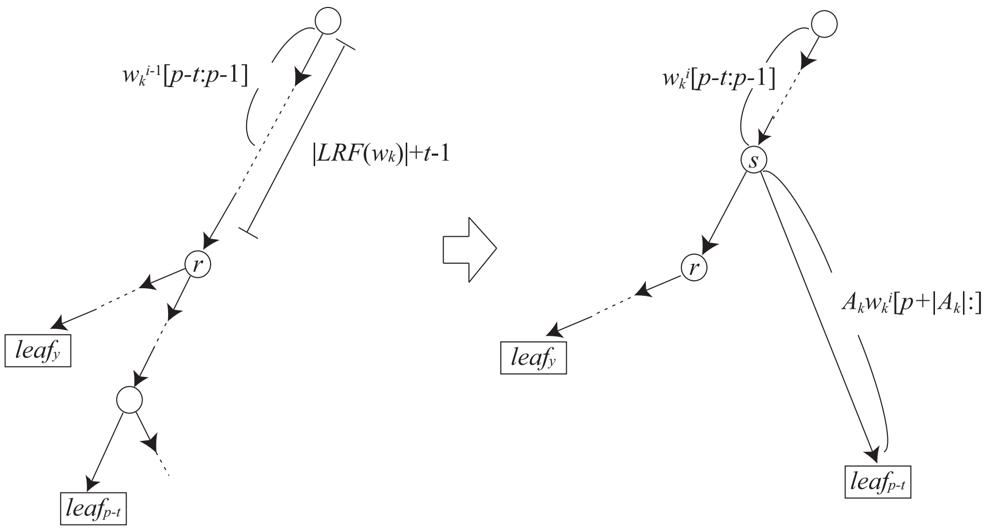
It is not difficult to see that the edge in each case of Lemma 7 does not exist in $SLSTree(w_k^{i-1})$. Hence we create the edge when we update $SLSTree(w_k^{i-1})$ to $SLSTree(w_k^i)$.

The next lemma states how to locate node s of Case 2 of Lemma 7.

Lemma 8 *For each t, we can locate node s such that $str(s) = w_k^i[p-t:p-1]$ in amortized constant time.*

Proof. Let x_{p-t} be the longest node in the tree such that $str(x_{p-t})$ is a prefix of $w_k^i[p-t:p-1]$.

Figure 5. Illustration of Case 2 of Lemma 7.


$w_k^{i-1}[p\text{-}t\text{:}p\text{-}1]$
$|LRF(w_k)|+t\text{-}1$
r
$leaf_y$
$leaf_{p-t}$
$w_k^{i}[p\text{-}t\text{:}p\text{-}1]$
s
r
$leaf_y$
$A_k w_k^i[p+|A_k|\text{:}]$
$leaf_{p-t}$

Consider the largest possible t and denote it by $t_{\max}$. Since $t_{\max} \le |LRF(w_k)|$, the node $x_{p-t_{\max}}$ can be found in $O(|LRF(w_k)|)$ time by going down the path that spells out $w_k^i[p-t_{\max} : p-1]$ from the root node (recall that Σ is fixed). Let $z \in \Sigma^*$ be the string such that $str(x_{p-t_{\max}})z = w_k^i[p-t_{\max} : p-1]$. If $z \neq \varepsilon$, then we create a new child node $s_{p-t_{\max}}$ of $x_{p-t_{\max}}$ such that $str(s_{p-t_{\max}}) = w_k^i[p-t_{\max} : p-1]$. Otherwise, we set $s_{p-t_{\max}} = x_{p-t_{\max}}$.

Now assume that we have located nodes x_{p-t} and s_{p-t}. We can then locate s_{p-t+1} as follows. Consider node x_{p-t+1}. Remark that $str(suf(x_{p-t}))$ is a prefix of $str(x_{p-t+1})$, and thus we can detect x_{p-t+1} in $O(|str(x_{p-t+1})| - |str(suf(x_{p-t}))|)$ time by using the suffix link. After finding x_{p-t+1}, we can locate or create s_{p-t+1} in constant time.

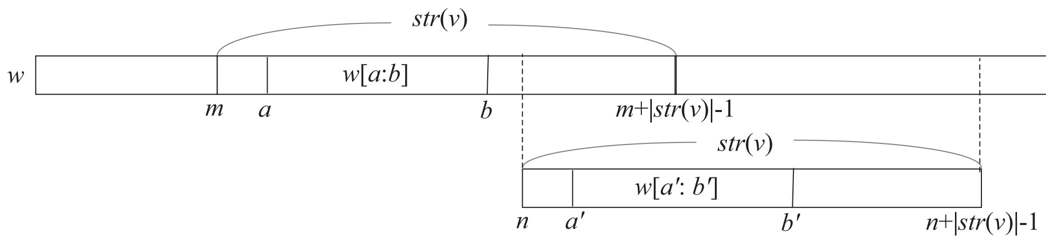
The total time cost for detecting x_{p-t} for all $1 \le t \le t_{\max}$ is linear in

$$\begin{aligned}
&\sum_{t=2}^{t_{\max}} (|str(x_{p-t+1})| - |str(suf(x_{p-t}))|) \\
&= |str(x_{p-1})| - |str(suf(x_{p-2}))| \\
&+ |str(x_{p-2})| - |str(suf(x_{p-3}))| \\
&\quad \ldots\ldots \\
&+ |str(x_{p-t_{\max}+1})| - |str(suf(x_{p-t_{\max}}))| \\
&= |str(x_{p-1})| - |str(suf(x_{p-t_{\max}}))| + t_{\max} - 2 \\
&= |str(x_{p-1})| - |str(x_{p-t_{\max}})| + t_{\max} - 1 \\
&\le t_{\max} \le |LRF(w_k)|.
\end{aligned}$$

Hence we can locate each s_{p-t} 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 $str(s) = w_k^i[p-t : p-1]$. In order to update $SLSTree(w_k^{i-1})$ to $SLSTree(w_k^i)$, 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 $\mathrm{card}(v) \ge 4$ in

Figure 6. Illustration of proof for Lemma 9.



$\mathit{SLSTree}(w_k^{i-1})$, 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 $\mathit{Dead}_{w_k^i}(y) = \mathit{BP}_{w_k^{i-1}}(y) \backslash \mathit{BP}_{w_k^i}(y)$, 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 $LRF(w_k)$ in w_k.

Lemma 9 *Let v be any reference node of $\mathit{SLSTree}(w_k^i)$ such that $\#\mathit{occ}_{w_k^i}(\mathit{str}(v)) = 1$. For any integer m, n, if $m, n \in \mathit{BP}_{w_k^i}(\mathit{str}(v))$, then there is no integer r such that $m < r < n$ and $r \in \mathit{Dead}_{w_k^i}(\mathit{str}(v))$. (See Figure 6).*

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

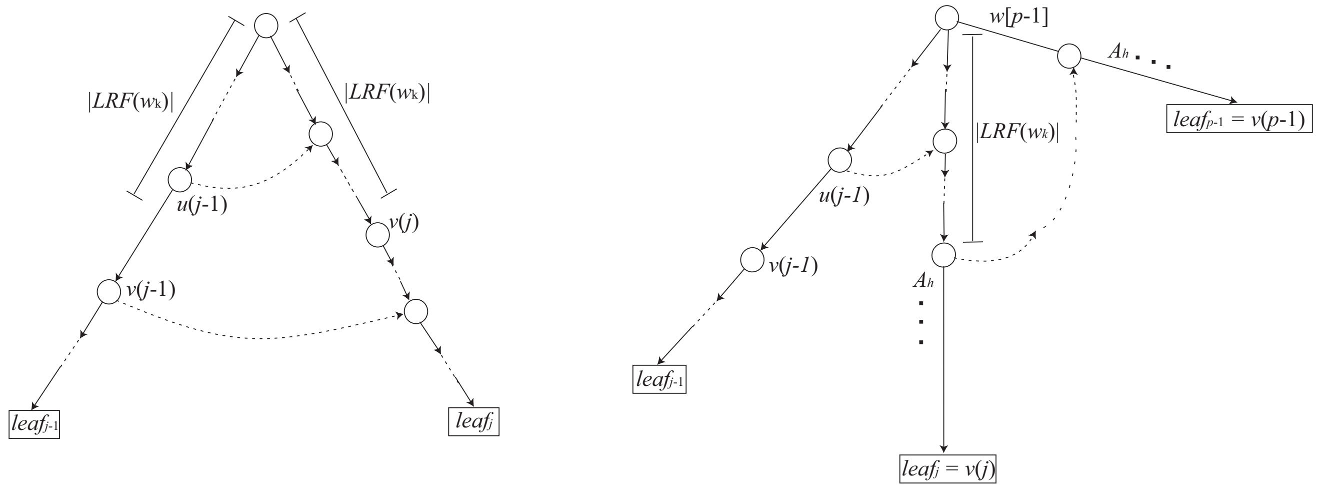
Recall that p is the beginning position of the i-th largest greedily selected occurrence of $LRF(w_k)$ in w_k. Also, for any $1 \leq t \leq |LRF(w_k)|$ such that $w_k^{i-1}[r] \in \Sigma$ for every $p - t \leq r < p$, we have removed leaf_{p-t} from the subtree rooted at the reference node v and have reconnected it to node s such that $\mathit{str}(s) = w_k^i[p - t : p - 1]$. According to the above lemma, if $\min(v) < p - t < \max(v)$, leaf_j for every $p - t \leq j \leq \max(v)$ 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 $0 \leq h \leq |LRF(w_k)| - 1$ has to be removed from the tree, since $w_k^i[p + h] \notin \Sigma$ and therefore this leaf node should not exist in $\mathit{SLSTree}(w_k^i)$ (see the dark-shaded suffixes of Figure 3). 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 10 *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 $p - |LRF(w_k)| \leq j \leq p + |LRF(w_k)| - 1$, we can locate the reference node $v(j)$ in amortized constant time.*

Figure 7. The left figure illustrates how to find $v(j)$ from $v(j-1)$. The right one illustrates a special case where $v(j) = leaf_j$. Once $v(j) = leaf_j$, it stands that $v(k) = leaf_k$ for any $j \leq k \leq p-1$.



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 $len(u(j-1)) \leq \ell$ and $len(v(j-1)) \leq \ell+1$. We go to $suf(u(j-1))$. Since $len(suf(u(j-1)))+1 = len(u(j-1))$, we have $len(suf(u(j-1))) \leq \ell+1$. Thus, we can find $v(j)$ by going down the path starting from $suf(u(j-1))$ and spelling out $w_k^{i-1}[j-1+len(u(j-1)) : j-1+len(v(j-1))] = w_k^{i-1}[j+len(suf(u(j-1))) : j-1+len(v(j-1))]$. (See also the left illustration of Figure 7).

A special case happens when there exists a node s in the path from the root to $leaf_j$, such that $len(s) = \ell$ 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 $v_j = leaf_j$. Moreover, for any $j \leq k \leq p-1$, $v(k) = leaf_k$. (See also the right illustration of Figure 7). It is thus clear that each $v(k)$ can be found in constant time. Since $|A_h| \geq \ell = LRF(w_k)$, the leaves corresponding to $w_k^{i-1}[p+x-1:]$ with $1 \leq x \leq \ell$ do not exist in $SLSTree(w_k^{i-1})$. □

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.

3.3. *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 $A_k \rightarrow x_k$ 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 in x_k. This way we can reduce

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 := $\emptyset$;
while true do
    while (n = bins.getNextOfLength(len)) = null do
        if len $\leq 2$ then return rules;
        foreach x $\in$ 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 $\geq$ n.pathlen /* n is repeating factor */ then
        nonTerm := new non-terminal symbol;
        rules := rules $\cup$ {nonTerm $\rightarrow$ 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 $\in$ 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:$\langle$min, max, card$\rangle$ = $\langle$pos, pos, 1$\rangle$; add edge (s, nonTerm, pos);
        if pos > occpos then w[pos] = $\bullet$; markDead(pos);
    w[occpos] := nonTerm; markDead(occpos);
return
```

Algorithm 3: **getGreedilySelectedOccurrences**

```
Input: LRFnode
Output: Set of greedily selected occurrences of LRFnode.path
1  gso := $\emptyset$;
2  foreach c $\in$ LRFnode.children do
3      occ := 0;
4      if notDead(c.min) then occ := c.min ;
5      else occ := find first occurrence of LRFnode.path after c.min+endOfDeadArea[c.min];
6      if occ $\neq 0$ && notDead(occ+LRFnode.pathlen$-1$) then
7          gso := gso $\cup$ {occ};
8          for pos = occ to occ+LRFnode.pathlen$-1$ do
9              markEndOfDeadArea(pos, occ+LRFnode.pathlen$-1$);
10         occ := occ + LRFnode.pathlen ;
11         if notDead(occ) && notDead(occ+LRFnode.pathlen$-1$) then gso := gso $\cup$ {occ};
12
13 return gso;
```

the total size of the grammar. In so doing, we consider an LRF of string $z_k = w_k\$_0x_1\$_1\cdots x_k\$_k$, 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 $LRF(z_0) = \texttt{abb}$. 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 $LRF(z_1) = \texttt{ab}$. 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 method of text compression *LFS2*.

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

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 $z_k = w_kx_1\$_1\cdots x_k\$_k$. 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 [29], 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 $O(|w|)$. □

4. Conclusions and Future Work

This paper introduced a linear-time algorithm to compress a given text by longest-first substitution (LFS). We employed a new data structure called sparse lazy suffix trees in the core of the algorithm.

We also gave a linear-time algorithm for LFS2 that achieves better compression than LFS.

A related open problem is the following: Does there exist a linear time algorithm for text compression by largest-area-first substitution (LAFS)? The algorithm presented in [21] uses *minimal augmented suffix trees* (*MASTrees*) [31] which enable us to efficiently find a factor of the largest area. The size of MASTrees is known to be linear in the input size [32], but the state-of-the-art algorithm of [32] to construct MASTrees takes $O(n \log n)$ time, where n is the input text length. Also, the algorithm of [21] for LAFS reconstructs the MASTree from scratch, every time a factor of the largest area is replaced by a new non-terminal symbol. Would it be possible to update a MASTree or its relaxed version for following substitutions?

Acknowledgments

We would like to thank Matthias Gallé and Pierre Peterlongo for leading us to reference [28].

References and Notes

1. Kida, T.; Matsumoto, T.; Shibata, Y.; Takeda, M.; Shinohara, A.; Arikawa, S. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science* **2003**, *298*, 253–272.
2. Mäkinen, V.; Ukkonen, E.; Navarro, G. Approximate Matching of Run-Length Compressed Strings. *Algorithmica* **2003**, *35*, 347–369.
3. Lifshits, Y. Processing Compressed Texts: A Tractability Border. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07)*. Springer-Verlag, 2007, Vol. 4580, *Lecture Notes in Computer Science*, pp. 228–240.
4. Matsubara, W.; Inenaga, S.; Ishino, A.; Shinohara, A.; Nakamura, T.; Hashimoto, K. Efficient Algorithms to Compute Compressed Longest Common Substrings and Compressed Palindromes. *Theoretical Computer Science* **2009**, *410*, 900–913.
5. Hermelin, D.; Landau, G. M.; Landau, S.; Weimann, O. A Unified Algorithm for Accelerating Edit-Distance Computation via Text-Compression. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS'09)*, 2009, pp. 529–540.
6. Matsubara, W.; Inenaga, S.; Shinohara, A. Testing Square-Freeness of Strings Compressed by Balanced Straight Line Program. In *Proc. 15th Computing: The Australasian Theory Symposium (CATS'09)*. Australian Computer Society, 2009, Vol. 94, *CRPIT*, pp. 19–28.
7. Nevill-Manning, C. G.; Witten, I. H. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artificial Intelligence Research* **1997**, *7*, 67–82.
8. Nevill-Manning, C. G.; Witten, I. H. Online and offline heuristics for inferring hierarchies of repetitions in sequences. *Proc. IEEE* **2000**, *88*, 1745–1755.
9. Giancarlo, R.; Scaturro, D.; Utro, F. Textual data compression in computational biology: a synopsis. *Bioinformatics* **2009**, *25*, 1575–1586.
10. Kieffer, J. C.; Yang, E.-H. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory* **2000**, *46*, 737–754.
11. Storer, J. NP-completeness Results Concerning Data Compression. Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

12. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory* **1978**, *24*, 530–536.
13. Welch, T. A. A Technique for High-Performance Data Compression. *IEEE Computer* **1984**, *17*, 8–19.
14. Kieffer, J. C.; Yang, E.-H.; Nelson, G. J.; Cosman, P. C. Universal lossless compression via multi-level pattern matching. *IEEE Transactions on Information Theory* **2000**, *46*, 1227–1245.
15. Sakamoto, H. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms* **2005**, *3*, 416–430.
16. Rytter, W. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science* **2003**, *302*, 211–222.
17. Sakamoto, H.; Maruyama, S.; Kida, T.; Shimozono, S. A Space-Saving Approximation Algorithm for Grammar-Based Compression. *IEICE Trans. on Information and Systems* **2009**, *E92-D*, 158–165.
18. Maruyama, S.; Tanaka, Y.; Sakamoto, H.; Takeda, M. Context-Sensitive Grammar Transform: Compression and Pattern Matching. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE'08)*. Springer-Verlag, 2008, Vol. 5280, *Lecture Notes in Computer Science*, pp. 27–38.
19. Wolff, J. G. An algorithm for the segmentation for an artificial language analogue. *British Journal of Psychology* **1975**, *66*, 79–90.
20. Larsson, N. J.; Moffat, A. Offline Dictionary-Based Compression. In *Proc. Data Compression Conference '99 (DCC'99)*. IEEE Computer Society, 1999, p. 296.
21. Apostolico, A.; Lonardi, S. Off-Line Compression by Greedy Textual Substitution. *Proc. IEEE* **2000**, *88*, 1733–1744.
22. Apostolico, A.; Lonardi, S. Compression of Biological Sequences by Greedy Off-Line Textual Substitution. In *Proc. Data Compression Conference '00 (DCC'00)*. IEEE Computer Society, 2000, pp. 143–152.
23. Ziv, J.; Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* **1977**, *IT-23*, 337–349.
24. Burrows, M.; Wheeler, D. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
25. Nakamura, R.; Bannai, H.; Inenaga, S.; Takeda, M. Simple Linear-Time Off-Line Text Compression by Longest-First Substitution. In *Proc. Data Compression Conference '07 (DCC'07)*. IEEE Computer Society, 2007, pp. 123–132.
26. Inenaga, S.; Funamoto, T.; Takeda, M.; Shinohara, A. Linear-time off-line text compression by longest-first substitution. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*. Springer-Verlag, 2003, Vol. 2857, *Lecture Notes in Computer Science*, pp. 137–152.
27. Bentley, J.; McIlroy, D. Data compression using long common strings. In *Proc. Data Compression Conference '99 (DCC'99)*. IEEE Computer Society, 1999, pp. 287–295.
28. Lanctot, J. K.; Li, M.; Yang, E.-H. Estimating DNA sequence entropy. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000, pp. 409–418.

29. Ukkonen, E. On-line Construction of Suffix Trees. *Algorithmica* **1995**, *14*, 249–260.
30. Kärkkäinen, J.; Ukkonen, E. Sparse Suffix Trees. In *Proc. 2nd Annual International Computing and Combinatorics Conference (COCOON'96)*. Springer-Verlag, 1996, Vol. 1090, *Lecture Notes in Computer Science*, pp. 219–230.
31. Apostolico, A.; Preparata, F. P. Data structures and algorithms for the string statistics problem. *Algorithmica* **1996**, *15*, 481–494.
32. Brødal, G. S.; Lyngsø, R. B.; Östlin, A.; Pedersen, C. N. S. Solving the String Stastistics Problem in Time $O(n \log n)$. In *Proc. 29th International Colloquium on Automata,Languages, and Programming (ICALP'02)*. Springer-Verlag, 2002, Vol. 2380, *Lecture Notes in Computer Science*, pp. 728–739.
33. Lanctot, J. K. *Some String Problems in Computational Biology*. PhD thesis, University of Waterloo, 2004.

Appendix

In this appendix we show that the algorithm of Lanctot et al. [28] for LFS2 takes $O(n^2)$ time, where n is the length of the input string.

Consider string

$$w = w_0 = z_0 = \texttt{aaaaaaabbbbbaaaabbbbbcaaaaaaaa\$}.$$

The Lanctot algorithm constructs a suffix tree of w, constructs a bin-sorted list of internal nodes of the tree, and updates the tree in a similar way to our algorithm in Section 3.3.. However, a critical difference is that any node v of their tree structure does *not* store 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 $\mathrm{card}(v) = |BP_w(str(v))|$.

See Figure 8 which illustrates the suffix tree of w.

A bin-sorted list of internal nodes of $STree(w)$ in decreasing order of their length is as follows:

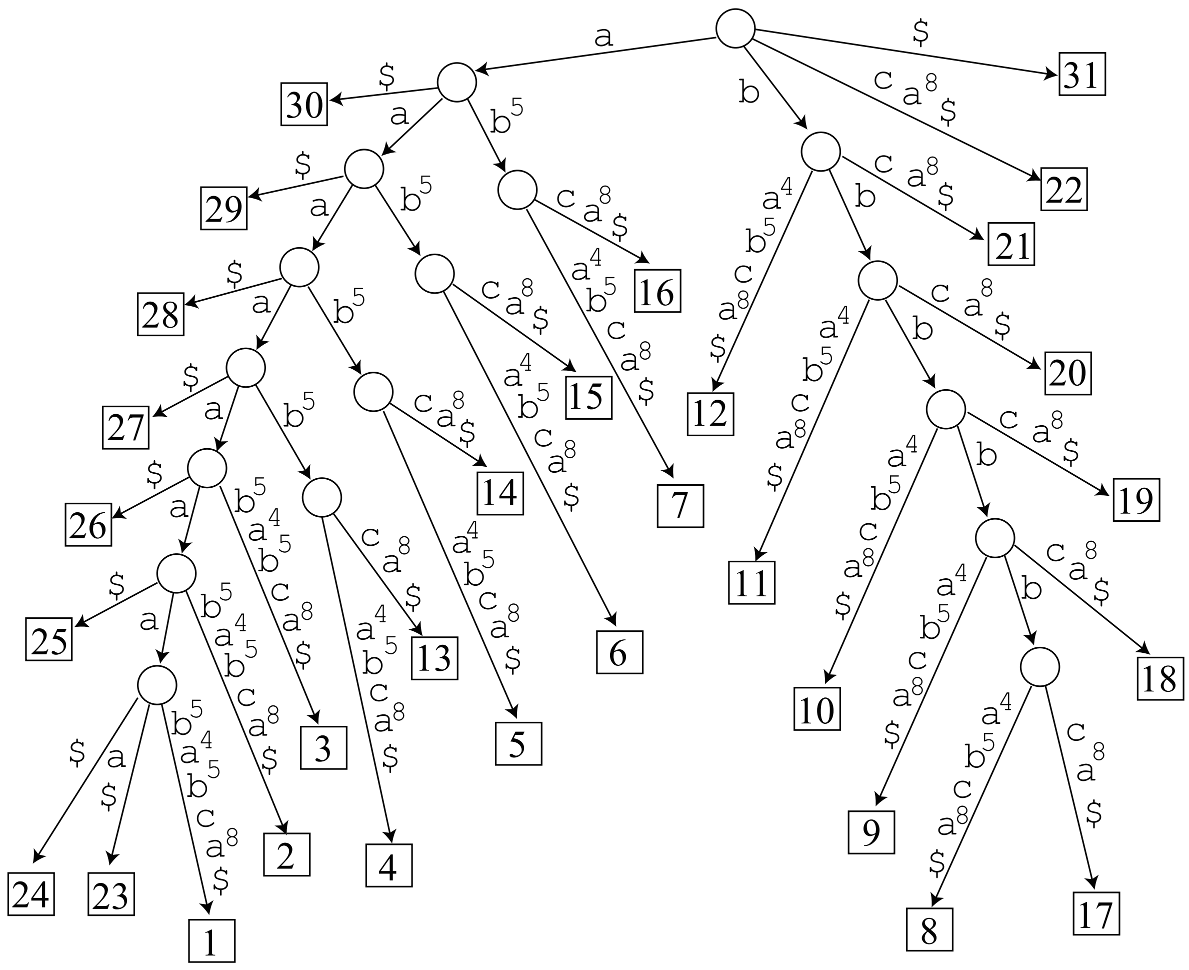
$$9 : \texttt{aaaabbbbb}$$
$$8 : \texttt{aaabbbbb}$$
$$7 : \texttt{aabbbbb}, \texttt{aaaaaaa}$$
$$6 : \texttt{abbbbb}, \texttt{aaaaaa}$$
$$5 : \texttt{bbbbb}, \texttt{aaaaa}$$
$$4 : \texttt{bbbb}, \texttt{aaaa}$$
$$3 : \texttt{bbb}, \texttt{aaa}$$
$$2 : \texttt{bb}, \texttt{aa}$$
$$1 : \texttt{b}, \texttt{a}$$

In [28], Lanctot et al. do not mention how they find occurrences of each node in the sorted list. Since they do not have an ordered triple $\langle \min(v), \max(v), \mathrm{card}(v) \rangle$ for each node v, the best possible way is to traverse the subtree of v checking the leaves in the subtree. Now, for the first LRF-candidate aaaabbbbb, we get positions 4 and 13 and find out that $LRF(w) = LRF(z_0) = \texttt{aaaabbbbb}$. Then we obtain

$$w_1 = \texttt{aaa}AA\texttt{caaaaaaaa\$},$$

where A is a new non-terminal symbol that replaces $LRF(z_0) = \texttt{aaaabbbbb}$.

Figure 8. $STree(w)$ with $w = \texttt{aaaaaaabbbbbaaaabbbbbcaaaaaaaa\$}$.



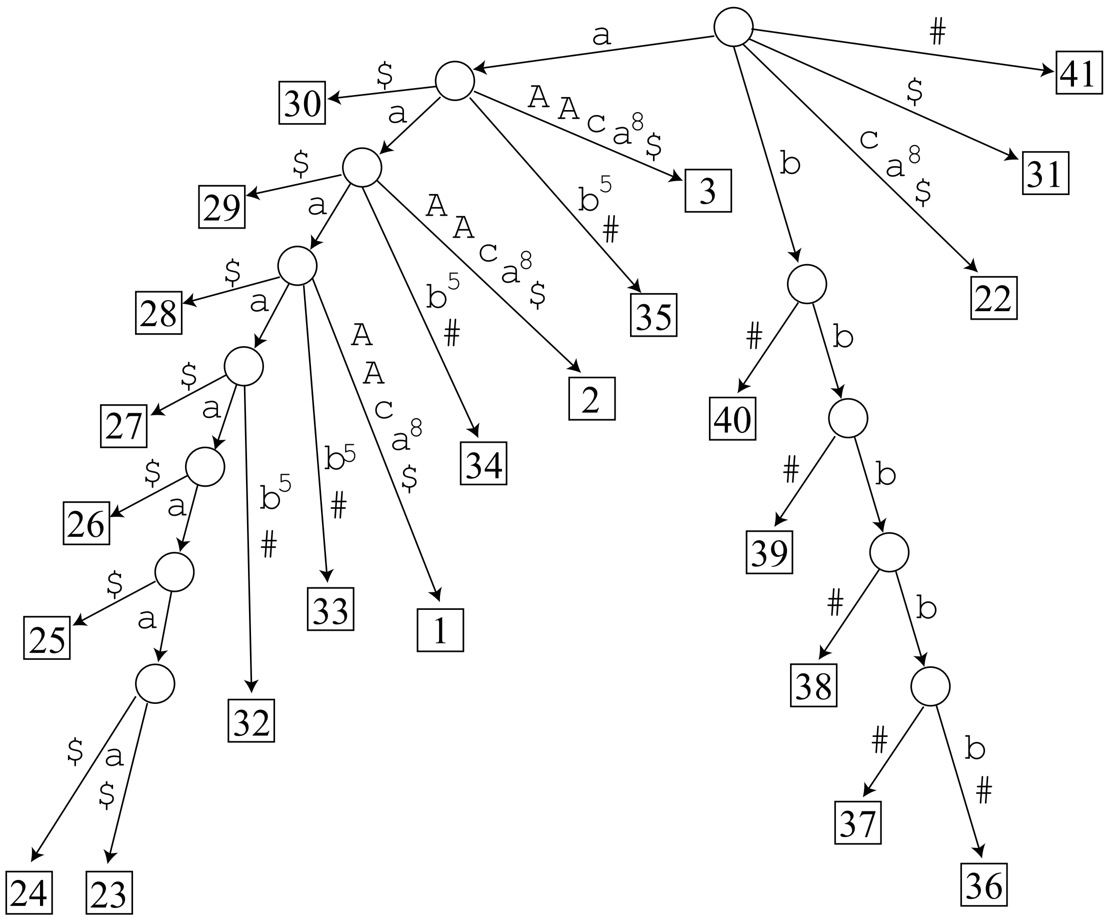
Now see Figure 9 which illustrates a generalized sparse suffix tree for

$$z_1 = \texttt{aaa}AA\texttt{caaaaaaaa\$aaaabbbbb}\#.$$

To find $LRF(z_1)$, we check the nodes in the list as follows.

- Length 8. The generalized suffix tree has no node representing aaabbbbb, and hence it is not an LRF.
- Length 7. Since node aaaaaaa exists in the generalized suffix tree, we traverse its subtree and find 2 occurrences 23 and 24 in z_1. However, it is not an LRF of z_1. The other candidate aabbbbb does not have a corresponding node in the tree, so it is not an LRF, either.
- Length 6. Node aaaaaa exists in the generalized suffix tree and we find 3 occurrences 23, 24 and 25 in z_1 by traversing the tree, but it is not an LRF. The tree has no node corresponding to abbbbb, hence it is not an LRF.
- Length 5. Node aaaaa exists in the generalized suffix tree and we find 4 occurrences 23, 24, 25 and 26 in z_1 by traversing the tree, but it is not an LRF. There is no node in the tree corresponding to bbbbb.

Figure 9. Generalized sparse suffix tree of $z_1 = \texttt{aaa}AA\texttt{caaaaaaaa}\$\texttt{aaaabbbbb}\#$.



- Length 4. Node aaaa exists in the generalized suffix tree and we find 5 occurrences 23, 24, 25, 26 and 27. Now 23 and 27 are non-overlapping occurrences of aaaa, and hence it is an LRF of z_1.

Focus on the above operations where we examined factors of lengths from 7 to 5. The total time cost to find the occurrences for the LRF-candidates of these lengths is proportional to 2 + 3 + 4, but *none of them is an LRF of* z_1 in the end.

In general, for any input string of the form

$$w = \texttt{a}^{2k-1}\texttt{b}^{k+1}\texttt{a}^{k}\texttt{b}^{k+1}\texttt{ca}^{2k}\$,$$

the time cost of the Lanctot algorithm for finding $LRF(z_1)$ is proportional to

$$2 + 3 + \cdots + k = \frac{(k-1)(k+2)}{2}.$$

Since $k = O(|w|) = O(n)$, the Lanctot algorithm takes $O(n^2)$ time.

In his PhD thesis [33], Lanctot modified the algorithm so that all the occurrences of each candidate factor in w are stored in each element of the bin-sorted list (Section 3.1.3, page 55, line 1). However, this clearly requires $O(n^2)$ space. Note that using a suffix array cannot immediately solve this, since the lexicographical ordering of the suffixes can change due to substitution of LRFs, and no efficient methods to edit suffix arrays for such a case are known.

On the contrary, as shown in Section 3., each node v of our data structure stores an ordered triple $\langle \min(v), \max(v), \mathrm{card}(v) \rangle$, and our algorithm properly maintains this information when the tree is updated. Using this triple, we can check in amortized constant time whether or not each node in the bin-sorted list is an LRF. Hence the total time cost remains $O(n)$.