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

Shunsuke Inenaga[1,2], Takashi Funamoto[1], Masayuki Takeda[1,2], and Ayumi Shinohara[1,2]

[1] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{s-ine, t-funa, ayumi, takeda}@i.kyushu-u.ac.jp
[2] PRESTO, Japan Science and Technology Corporation (JST)

Abstract. Given a text, grammar-based compression is to construct a grammar that generates the text. There are many kinds of text compression techniques of this type. Each compression scheme is categorized as being either *off-line* or *on-line*, according to how a text is processed. One representative tactics for off-line compression is to substitute the *longest* repeated factors of a text with a production rule. In this paper, we present an algorithm that compresses a text basing on this longest-first principle, in linear time. The algorithm employs a suitable index structure for a text, and involves technically efficient operations on the structure.

1 Introduction

Text compression is one of the main stream in the area of string processing [4]. The aim of compression is to reduce the size of a given text by efficiently removing the redundancy of the text. Compressing a text enables us to save not only memory space for storage, but also time for transferring the text since its compressed size is now smaller. It is ideal to compress the text as much as possible, but compression in reality has to be done in the trade-off between time and space, i.e. text compression algorithms are also required to have fast performance.

One major scheme of text compression is *grammar-based* text compression, where a grammar that produces the text is generated. Many attempts to generate a smaller grammar have been made so far, such as in the well-known LZ78 algorithm [20] and the SEQUITUR algorithm [14, 15]. These two algorithms both process an input text *on-line*, namely, they read the text in a single pass, and begin to emit compressed output (production rules for a grammar) before they have seen all of the input. Actually, the history of text compression algorithms began with processing texts on-line, since limitation of available memory space has until recently been a big concern. On-line algorithms run on relatively small space by employing the idea of a sliding window, but they only generate a grammar based on replacing the repeating factors in the window that is of bounded size. Therefore some possibilities to compress texts into smaller sizes would remain.

M.A. Nascimento, E.S. de Moura, A.L. Oliveira (Eds.): SPIRE 2003, LNCS 2857, pp. 137–152, 2003.

Due to recent hardware developments, we are now allowed to dedicate more memory space to text compression. This gives us opportunities to design *off-line* algorithms that more efficiently process an input text and give us better compression. Two strategies for seeking for repeating factors in the whole input text are possible; the *most-frequent-first* and *longest-first* strategies.

Text compression by the most-frequent-first substitution was first considered by Wolff [19]. His algorithm is, given a text, to recursively replace the most frequently occurring digram (factor of length two) with a new character, which results in a production rule corresponding the digram. Though Wolff's algorithm takes $O(n^2)$ time for an input text of length n, Larsson and Moffat [12] devised a clever algorithm, named Re-Pair, that runs in $O(n)$ time and compresses the text by recursively substituting new characters for the most frequent digram.

In this paper we consider the other one, text compression by the longest first substitution, where we generate a grammar by substituting new characters for the longest repeating factors of a given text of length more than one. For example, from string `abcacaabaaabcacbabababcaccabacabcac` of length 35 we obtain the following grammar

$$S \rightarrow A\mathtt{a}B\mathtt{a}A\mathtt{b}B\mathtt{b}A\mathtt{c}B\mathtt{c}A$$
$$A \rightarrow \mathtt{abcac}$$
$$B \rightarrow \mathtt{aba}.$$

of size 24. Bentley and McIlroy [5] gave an algorithm for this compression scheme, but Nevill-Manning and Witten [16] 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. This paper, therefor, introduces the first explicit, and complete, linear-time algorithm for text compression with the longest-first substitution. The core of our algorithm is the use of *suffix trees* [18], for they are quite useful for finding the longest repeating factors as is mentioned in [16]. Our algorithm, which is really combinatorial, involves highly technical but necessary update operations on suffix trees towards upcoming substitutions. We give a precise analysis for the time complexity of our algorithm, which results in being linear in the length of an input text string.

2 Preliminaries

2.1 Notations on Strings

Let Σ be a finite alphabet. 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 sets of all prefixes, factors, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Factor}(w)$, and $\mathit{Suffix}(w)$, respectively.

The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The i-th character 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 factor x of a string w, let $BegPos_w(x)$ denote the set of the beginning positions of all occurrences of x in w.

For a non-empty factor x of a string w, $\#occ_w(x)$ denotes 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 $x \in \Sigma^+$. An integer $1 \leq p \leq |x|$ is said to be a *period* of x if the suffix $x[p+1:]$ of x is also a prefix of x, that is, $x[p+1:] = x[1:|x|-p]$.

2.2 Suffix Trees

The *suffix tree* of a string w, denoted by $STree(w)$, is an efficient index structure which is defined as follows:

Definition 1. *$STree(w)$ 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 characters;*
4. *every suffix of w is spelled out in a path starting from the root node.*

Quite a lot of applications of suffix trees have been introduced so far, in the literature such as [1, 9, 8].

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 $STree(w)$. $STree(w)$ for string `ababa$` is shown in Fig. 1. For any node v of $STree(w)$, $label(v)$ 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 $length(v)$, is defined to be $|label(v)|$. The *number* of the leaf node of $STree(w)$ corresponding to $w[i:]$ is defined to be i, for $1 \leq i \leq |w|$. The i-th leaf node of $STree(w)$ is denoted by $leaf_i$. Every node v of $STree(w)$ except for the root node has the *suffix link*, denoted $suf(v)$, such that $suf(v) = v'$ where $label(v') \in Suffix(label(v))$ and $length(v') + 1 = length(v)$.

If there exists a node v in $STree(w)$ such that $label(v) = x$ for some $x \in Factor(w)$, then we sometimes specify that x is represented by an *explicit* node. Otherwise, we say that x is represented by an *implicit* node in $STree(w)$. The implicit node is indicated by a *reference* pair $\langle s, \alpha \rangle$ of a node and string, such that $label(s) \cdot \alpha = x$.

Actually, every edge label x of $STree(w)$ is implemented by a pair $\langle i, j \rangle$ of integers such that $x = w[i:j]$, and thus occupies only constant space. Therefore, the size of $STree(w)$ is linear in $|w|$. More precisely:

Theorem 1 (McCreight [13]). *For any string $w \in \Sigma^*$ with $|w| > 1$, $STree(w)$ has at most $2|w| - 1$ nodes and $2|w| - 2$ edges.*

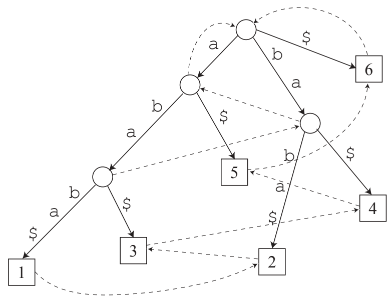

Fig. 1. *STree*(w) with $w = \mathtt{ababa}\$$. Solid arrows represent edges, and dotted arrows are suffix links.

Moreover, on the assumption that Σ is fixed;

Theorem 2 (Weiner [18]). *For any string $w \in \Sigma^*$, STree(w) can be constructed in linear time.*

Construction of *STree*(w) has been studied in various contexts. For instance, Weiner [18] gave the first algorithm to construct *STree*(w) in linear time. Later on, McCreight [13] and Ukkonen [17] individually presented conceptionally new linear-time algorithms for construction of *STree*(w). A merit of the two latter algorithms is that the order of the creation of a leaf node exactly corresponds to the beginning position of the suffix represented by the leaf node. Namely, the i-th created leaf node of *STree*(w) is exactly leaf_i for any $1 \leq i \leq |w|$. Hereby we can easily associate each leaf node with its number, without any extra effort after the construction of *STree*(w) is completed.

3 Off-Line Compression by Longest-First Substitution

Given a text string $w \in \Sigma^*$, we here consider to replace an LRF x of w such that $|x| \geq 2$, with a new character not appearing in w. We call this operation *longest-first substitution* on w. Applying it to w as many times as possible, we can accomplish encoding of w, where we resultingly obtain a grammar consisting of the rules that produce the replaced factors. For instance, let us consider string `abaaabbababb$`, which has two LRFs `aba` and `abb`. Let us here choose `abb` for being replaced by a new character A, and then we obtain

$$S \to \mathtt{abaa}A\mathtt{ab}A\$$$
$$A \to \mathtt{abb}.$$

Replacing `ab` by B results in a grammar consisting of the production rules

$$S \to B\mathtt{aa}ABA\$$$
$$A \to \mathtt{abb}$$
$$B \to \mathtt{ab}.$$

3.1 Suffix Trees Are Useful for Longest-First Substitution

To compress w according to the above principle and in $O(|w|)$ time, we need to find in *(amortized) constant time* an LRF of w at every stage of compression. Preprocessing w is a direct and clever choice for this purpose, and concretely, we first construct $STree(w)$. We consider only the strings corresponding to the internal nodes of $STree(w)$ as candidates for LRFs. Since there can be LRFs of w that are not represented as nodes of $STree(w)$, one may think that such LRFs remain unsubstituted for, and violate our longest-first principle (e.g., see $STree(\texttt{ababa\$})$ of Fig. 1 in which factor `ab` is an LRF of `ababa$`, but is represented only as an implicit node.). However, we can fortunately prove the following lemma which guarantees that we have only to consider the strings represented as an internal node of $STree(w)$. This lemma is essential to our algorithm for text compression with longest-first substitution.

Lemma 1. *Suppose x is an LRF of w not corresponding to a node of $STree(w)$. Then, there exists another LRF y of w that corresponds to an internal node such that $|x| = |y|$ and $\#occ_w(y) \geq \#occ_w(x) = 2$. Moreover, x is no longer present in the string after the substitution for y. (See Fig. 2.)*

Proof. Suppose the implicit node representing x is on the edge from some node s to node t of $STree(w)$. Let $u = label(t)$, and then we have $BegPos_w(x) = BegPos_w(u)$. Since x is an LRF of w and a proper prefix of u, the string u is not repeating. Let i, j be the minimum and maximum elements of $BegPos_w(u)$, respectively. It is obvious that $j - i = |x| < |u|$ and therefore the string u has a period $|x|$. Let $\ell = |u|$. The string $w[i : j + \ell - 1] = xu$ has a period $|x|$. Let p be the smallest period of the same string, and let z be the length-p prefix of x. By the periodicity lemma, we can show that $x = z^k$ for some $k \geq 1$ as in Fig.2. Let ℓ' ($\ell' \geq \ell$) be the largest integer such that the string $w[i : j + \ell' - 1]$ has a period p. It is not hard to show that $w[i : j + \ell' - 1] = z^{2k}z'$ for some prefix z' of z. Let y be the length-$|x|$ suffix of this string, and y' be the length-$|z'|$ prefix of x. Then, $w[i : j + \ell' - 1] = y'yy$. Let $a = w[j + \ell']$ and $b = w[j + \ell' - p]$. From the choice of ℓ', the characters a, b must be distinct. Since $|y| = k \cdot p$, we have $b = w[j + \ell' - p] = w[j + \ell' - |y|]$. The occurrences of y at positions $j + \ell'$ and $j + \ell' - |y|$ in w are followed by a and b, respectively, and therefore y is represented as an explicit node of $STree(w)$. Since x occurs only within the region $w[i : j + \ell' - 1]$, it cannot be present after substitution for the occurrences of y. $\square$

The above lemma implies that it suffices to consider the strings corresponding to the internal nodes of $STree(w)$ as candidate repeating factors for substitution. In fact, we only need to consider the LRF `ba` of `ababa$` that is represented by an explicit internal node of $STree(\texttt{ababa\$})$ of Fig. 1, in spite of the implicit one `ab` By sorting the internal nodes of $STree(w)$ in the order of their path lengths, we can maintain the list of such candidates. Notice that, however, the above lemma does not address every node of $STree(w)$ corresponds to a repeating factor of w. Namely, an overlapping factor x with $\#occ_w(x) = 1$ may be represented by

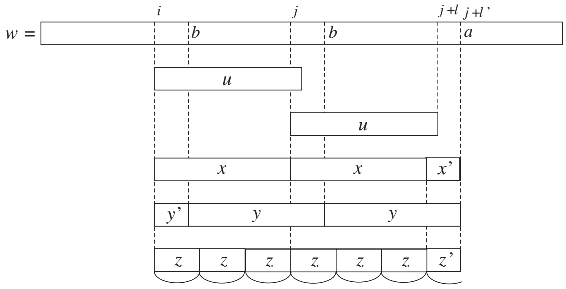

Fig. 2. An illustration for Lemma 1. An LRF x of w not corresponding to a node of $STree(w)$ implies two consecutive occurrences of x. In this case, there necessarily exists an LRF y corresponding to an internal node. The replacement of the two consecutive occurrences of y destroys the occurrences of x.

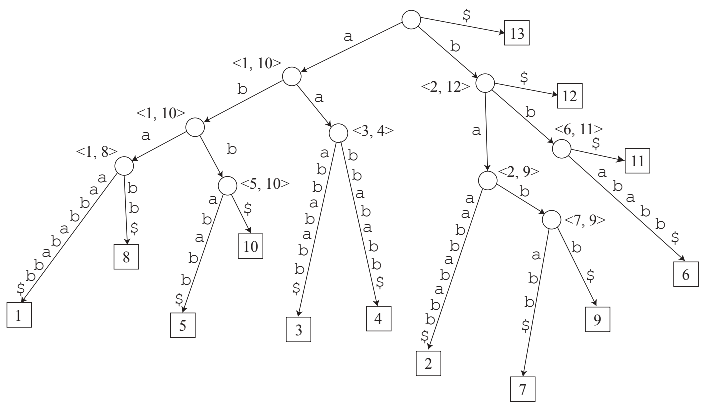

Fig. 3. Every node v of $STree(\texttt{abaaabbababb\$})$ shown here has got a pair $\langle i, j \rangle$, where the leftmost and rightmost occurrences of $label(v)$ are i, j, respectively.

a node of $STree(w)$. For example, see Fig. 1 displaying $STree(\texttt{ababa\$})$. Remark factor `aba` appears twice in the string, but $\#occ_u(\texttt{aba}) = 1$ since the two occurrences are overlapping. Let i, j be the beginning positions of the leftmost and rightmost occurrences of a factor x of a string w, respectively. If $|x| > j - i$, then it means that all occurrences of x are overlapping in w, and thus $\#occ_w(x) = 1$. Otherwise, we have $\#occ_w(x) \geq 2$, and therefore string x is a repeating factor of w. For any internal node s of $STree(w)$, the beginning position of the leftmost (rightmost) occurrence of $label(s)$ can be computed by a standard bottom-up traversal of the tree issuing the numbers of the leaf nodes upward. The time cost is proportional to the number of the edges in $STree(w)$, which is $O(|w|)$.

See Fig. 3, where every node v of $STree(\texttt{abaaabbababb\$})$ has got a pair $\langle i, j \rangle$ of integers, where i, j are the beginning positions of the leftmost and rightmost occurrences of $label(v)$, respectively.

```
1   5    10   15   20   25   30
abaabaabaabaababaabaabababaabaaba$
```

Fig. 4. An example for a string in which some occurrences of its LRF are overlapping.

The sole remaining matter is how to construct the list of the internal nodes for substitutions, which has to be sorted by the lengths of the nodes. It can simply be done by a bin sort in linear time in the number of internal nodes in $STree(w)$, therefore in $O(|w|)$ time (according to Theorem 1).

As a result of the above discussion, it has been shown that $STree(w)$ is quite effective in providing us the list of the repeating factors of w sorted in the decreasing order of their lengths. In the following sections we will see how an LRF of w is actually replaced by a new character, and what maintenance has to be done for the suffix tree.

3.2 Substitution for Longest Repeating Factor

According to the discussion in the previous section, we have got the list of nodes candidate for longest first substitution, and now the first element of the list corresponds to an LRF x of w. If $|x| < 2$, then any substitution does not reduce the size of the string, and thus we halt here. Otherwise, we actually replace x with a new character, say A, and then create the production rule $A \to x$.

A subtle consideration reveals that every occurrence of an LRF x in w is *not* allowed to be replaced by A, if w contains some overlapping occurrences of x. Conversely, we then could have more than one choice of the occurrences of x for being replaced by A. See Fig. 4 in which the string shown contains `abaabaaba` as a unique LRF. For example, we can choose the occurrences of `abaabaaba` beginning at positions 7 and 25 for substitution. Then, no other occurrences of `abaabaaba` cannot be replaced since they are overlapping either of the two chosen occurrences. Notice, however, we have $\#occ_u(\texttt{abaabaaba}) = 3$, that is, the occurrences beginning at positions 1, 15 and 25 could be chosen to be replaced, for instance. Below we give a way to choose exactly $\#occ_w(x)$ occurrences of an LRF x of a string w for substitution.

Definition 2. *Let x be a non-empty factor of $w \in \Sigma^*$. The* left-first greedily selected occurrences *of x in w is the sequence $i_1, \ldots, i_k$ ($k \geq 1$) of integers satisfying:*

1. *$i_1 = \min BegPos_w(x)$.*
2. *i_ℓ is the smallest integer such that $i_\ell \in BegPos_w(x)$ and $i_{\ell-1} + |x| \leq i_\ell$, for every $\ell = 2, \ldots, k$.*
3. *There is no integer i such that $i \in BegPos_w(x)$ and $i_k + |x| \leq i$.*

Proposition 1. *Let x be a non-empty factor of $w \in \Sigma^*$. If $i_1, \ldots, i_k$ ($k \geq 1$) is the left-first greedily selected occurrences of x in w, then $k = \#occ_w(x)$.*

The above proposition states that the left-first greedy choice of occurrences of an LRF for substitutions achieves the maximum number of substitutions. What has to be considered next is how to sort the positions of occurrences of an LRF in the increasing order.

The proposition below follows from the periodicity lemma.

Proposition 2. *For any non-empty factor x of a string w and integer ℓ with $1 \leq \ell \leq |w|$, the set $S = \{i \mid i \leq \ell \leq i + |x|$ and $x = w[i : i + |x| - 1]\}$ forms a single arithmetic progression. If $|S| \geq 3$, then the step is the smallest period of x. All the occurrences of x at positions $i \in S$ with $i \neq \max S$ are followed by a unique character.*

Lemma 2. *For any non-repeating factor x of w, the set $BegPos_w(x)$ forms a single arithmetic progression. When $|BegPos_w(x)| \geq 3$, the step is the smallest period of x.*

Proof. Let ℓ be the maximum element of $BegPos_w(x)$. Since x is non-repeating, $i \leq \ell \leq i + |x|$ for every $i \in BegPos_w(x)$. We can apply Proposition 2 to prove the lemma. □

Remark that an arithmetic progression can be represented as a triple of the first and last elements, and the number of its elements. We store in every internal node s of $STree(w)$ the triple of the minimum element, the maximum element, and the cardinality of $BegPos_w(u)$, which is a compact representation of the set $BegPos_w(u)$ if u is non-repeating, where $u = label(s)$.

The next proposition directly follows from the definition of $BegPos$.

Proposition 3. *Let s be an internal node of $STree(w)$ having children $s_1, \ldots, s_k$. Then, the set $BegPos_w(label(s))$ is the disjoint union of the sets*

$$BegPos_w(label(s_1)), \ldots, BegPos_w(label(s_k)).$$

Lemma 3. *Suppose x is an LRF of w corresponding to an internal node s of $STree(w)$. Let $s_1, \ldots, s_k$ be the children of s. Then, $BegPos_w(x)$ is the disjoint union of $BegPos_w(label(s_1)), \ldots, BegPos_w(label(s_k))$, each of which forms a single arithmetic progression.*

Proof. Notice that the strings $label(s_1), \ldots, label(s_k)$ are non-repeating because they are longer than x that is an LRF of w. We can prove the lemma by Proposition 3 and Lemma 2. □

For finite sets S, T of integers, we write $S \prec T$ if every element of S is smaller than any of T.

Lemma 4. *Suppose x is an LRF of w corresponding to an internal node s of $STree(w)$. Let $s_1, \ldots, s_k$ be the children of s arranged in the increasing order of the minimum elements of $BegPos_w(label(s_i))$. Then,*

$$BegPos_w(label(s_1)) \prec \cdots \prec BegPos_w(label(s_k)).$$

Proof. It suffices to prove the next claim.

Claim. For any child t of s with $|BegPos_w(label(t))| \leq 2$, the node t has no sibling t' such that $BegPos_w(label(t'))$ contains an integer k with $i < k < j$, where i and j are the minimum and maximum elements of $BegPos_w(label(t))$.

Let $u = label(t)$ and let x' be the prefix of u of length $j-i$. Since x is an LRF of w and x' is repeating, x cannot be shorter than x' and thus we have $|x| \geq j-i$. Assume, for a contradiction, that t has a sibling t' such that $BegPos_w(label(t'))$ contains an integer k with $i < k < j$. Since j belongs to the intervals $[i, i+|x|]$, $[k, k+|x|]$, and $[j, j+|x|]$, we can show that $\{i, k, j\}$ is a subset of an arithmetic progression and $w[i+|x|] = w[k+|x|]$ by Proposition 2. On the other hand, the characters $w[i+|x|]$ and $w[k+|x|]$ are the first characters of the labels of the edges from s to t and t', respectively. Hence the two characters must be distinct, a contradiction. The proof of the claim is now complete. □

The above lemma implies that we have only to sort the k integers that are, respectively, the minimum elements of $BegPos_w(label(s_1)), \ldots, BegPos_w(label(s_k))$. The discussion below, however, reveals that we indeed need not explicitly sort these k integers.

Recall that an edge label α in the suffix tree of a string w is represented by an ordered pair $\langle i, j \rangle$ of integers with $w[i:j] = \alpha$.

Proposition 4. *Ukkonen's suffix-tree construction algorithm guarantees that the first argument i of the ordered pair representing the label of the edge from a node s to a node t in $STree(w)$ is equal to* $\min BegPos_w(label(t)) + |label(s)|$.

The above proposition states that it suffices to arrange the out-going edges of a node s in the increasing order of the first arguments of the corresponding pairs. A short consideration reveals that this order coincides with the order of creation of the edges by Ukkonen's algorithm. Thus, all we have to do is to keep, for every node s, the list of the out-going edges of s arranged in the order of creation, which can be easily done during the suffix tree construction.

Finally, we achieve the following lemma.

Lemma 5. *For any LRF x corresponding to an internal node s of $STree(w)$ of a string w, the left-first greedily selected occurrences of x in w can be enumerated in $O(k)$ time, after an $O(|w|)$ time and space preprocessing of w, where k is the number of children of s.*

Proof. It is feasible in $O(|w|)$ time and space to build $STree(w)$ and store in each node t the triple of the minimum element, the maximum element, and the cardinality of $BegPos_w(label(t))$. By Lemma 3 and Lemma 4, we can prove the lemma. □

3.3 Preparation for Next Substitution

In this section, we show how to maintain our suffix-tree based data structure after the substitution for an LRF of a string w, in order to prepare for the next LRF substitution. Let x_k denote the string being replaced with a new character, say A_k, at the k-th stage of the compression of string w with longest-first substitution. Let $w_1 = w$, and let w_{k+1} denote the string obtained by replacing every occurrence of x_k in w_k that is greedily selected in the left-first manner, with A_k which is followed by $(|x_k| - 1)$-times repetition of a special character $\bullet \notin \Sigma$. The aim of the introduction of the special character $\bullet$ is so that we have $|w_k| = |w|$ for every k. The string obtained by removing all $\bullet$'s from w_k, is denote by $\overline{w_k}$. Clearly, $\overline{w_k}$ is identical to the string obtained just after the $(k-1)$-th stage of the compression of w. By definition, x_k is an LRF of $\overline{w_k}$.

Proposition 5. *For every k, the string x_k consists only of characters from Σ.*

Proof. Assume contrarily that x_k contains a character A_j for some $j < k$, with which some occurrences of x_j have been replaced since the j-th stage. Because $\#occ_{w_k}(x_k) \geq 2$, we have $\#occ_{w_j}(x_k) \geq 2$. This implies that x_k is a longer repeating factor of $\overline{w_j}$ than x_j, and this is a contradiction. $\square$

We say that a position i of w_k $(1 \leq i \leq |w|)$ is *active* if $w_k[i] \in \Sigma$, and *inactive*, otherwise. Let Act_k and $Inact_k$ be the sets of the active positions and inactive positions of w_k, respectively, for every k. $Act_1 = \{1, \ldots, |w|\}$ and $Inact_k = \emptyset$ as $w_1 = w$. Due to Proposition 5, we have $Act_1 \supset Act_2 \supset \cdots$.

In the running example with $\mathtt{abaaabbababb\$}$, the sequence $\mathtt{abaa}A\bullet\bullet\mathtt{ab}A\bullet\bullet\$$ is yielded after the substitution of A for the LRF $\mathtt{abb}$, where every position assigned $\bullet$ or A is now inactive. We now have $Act_2 = \{1, 2, 3, 4, 8, 9, 13\}$ and $InAct_2 = \{5, 6, 7, 10, 11, 12\}$. After the substitution of B for the next LRF $\mathtt{ab}$, the sequence $B \bullet \mathtt{aa}A \bullet \bullet B \bullet A \bullet \bullet\$$ is yielded, which gives us $Act_3 = \{3, 4, 13\}$ and $Inact_3 = \{1, 2, 5, 6, 7, 8, 9, 10, 11, 12\}$.

The data structure we want to maintain for $k = 1, 2, \ldots$ resembles the *sparse suffix tree* [11] of w_k that represent only the suffixes beginning at the active positions of w_k. In the sequel, we present an update procedure for this data structure. It is obvious that the following lemma stands.

Lemma 6. *For any factor y of w_{k+1} with $y \in \Sigma^+$, $\#occ_{w_{k+1}}(y) < \#occ_{w_k}(y)$ if and only if an occurrence of y overlaps some occurrence of x_k in w_k.*

See Fig. 5, in which an LRF x_k beginning at position i of w_k is being replaced by a new character A. First we consider a suffix of w_k beginning at position j with $j \leq i$. The latter part of such a suffix after position i has to be modified, since its factor x_k is converted to A at position i. The number of such suffixes is proportional to i, and thus it reaches $O(|w_k|)$ in the worst case. However, the suffixes we actually have to care are only those beginning at position j with $i - |x_k| + 1 \leq j \leq i$, since in the principle of the longest-first substitution any LRF x_{k+1} cannot be longer than x_k, and all we need to know is if $\#occ_{w_{k+1}}(x_{k+1})$ becomes smaller than $\#occ_{w_k}(x_{k+1})$ and it only happens if x_{k+1} overlaps x_k in w_k (by Lemma 6). Hereby we define the *attentional zone* for x_k with respect to position i to be the region from $i - |x_k| + 1$ to i. In the right figure of Fig. 5, the suffixes in the attentional zone are light shaded.

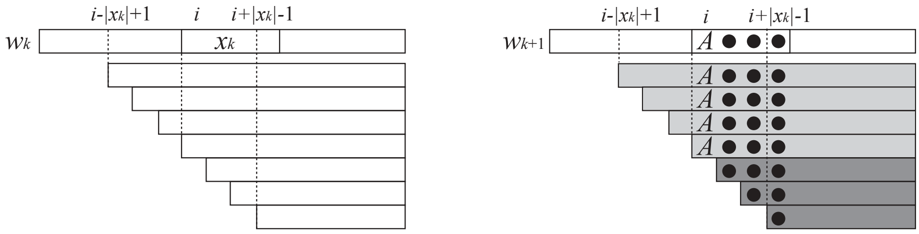

Fig. 5. Changes of the suffixes affected by the replacement of the occurrence of x_k beginning at position i of w_k during the k-th stage (from the left figure into the right figure). The occurrence of x_k in w_k is replaced with A followed by $(|x_k|-1)$-times repetition of $\bullet$ in w_{k+1}. In the right figure, the light-shaded region and the dark-shaded region denote the attentional and dead zones, respectively. The suffixes of w_k beginning at the positions in the attentional zone are modified accordingly, and those in the dead zone are no longer present in the sparse suffix tree for w_{k+1}.

To update our data structure for w_k to that for w_{k+1} according to the substitution for the LRF x_k, we have to check all the paths corresponding to the suffixes beginning at the positions in the attentional zone, and convert each of them accordingly. If naively traversing all these paths from the root node of the tree, then the total time cost will be $O(\#occ_{w_k}(x_k) \times |x_k|^2)$. However, we have the following lemma that reduces it to linear time.

Lemma 7. *At every k-th stage it is feasible in $O(|x_k|)$ time to maintain all paths spelling out a suffix of w_k which begins at a position in the attentional zone of w_k.*

Proof. Let $j = i-|x_k|+1$, and $u_j = w_k[j : i-1], u_{j+1} = w_k[j+1 : i-1], \ldots, u_i = w_k[i : i-1] = \varepsilon$. Note any position in the attentional zone is in Act_k. Let s_j and t_1 be the longest nodes in the tree for w_k, such that $label(s_j) \in Prefix(u_j)$ and $label(t_j) \in Prefix(u_j x_k)$, respectively (see the left figure of Fig. 6). Note $label(s_j)$ is a prefix of $label(t_j)$. These two nodes can be found by simply traversing the path spelling out $u_j x_k$ from the root node of the tree. Since $|u_j| + 1 = |x_k|$, the traversal can be done in $O(|x_k|)$ time (assuming $|\Sigma|$ is constant). Let $z \in \Sigma^*$ be the string such that $label(s_j) \cdot z = u_j$. If $z \neq \varepsilon$, then we create a new child node v_j of s_j such that $label(v_j) = u_j$. Otherwise, suppose $v_j = s_j$. Note that node t_j always has a unique out-going edge that is in the path spelling out $u_j x_k$ from the root node. Let r_j be the child node of t_j connected by this edge, and let y_j be the label of this edge. We reconnect r_j to v_j with the edge labeled by $A_k y$, and then remove the out-going edge of v_j which no longer has a node underneath (see the right figure of Fig. 6). This operation takes only constant time.

Now we focus on u_ℓ for some $j < \ell \leq i$. We need to find where nodes s_ℓ and t_ℓ in the tree such that $label(s_\ell) \in Prefix(u_\ell)$ and $label(t_\ell) \in Prefix(u_\ell)$, respectively. Remark that we have $label(suf(s_{\ell-1})) \in Prefix(s_\ell)$ and $label(suf(t_{\ell-1})) \in$

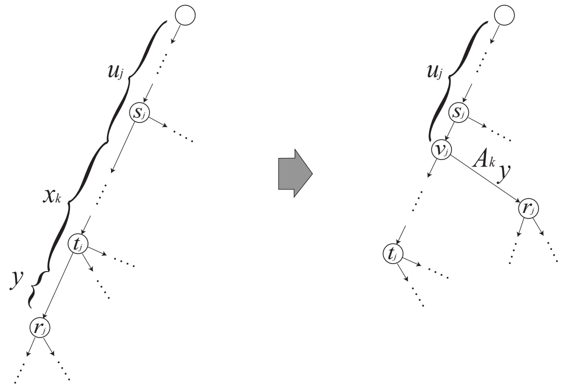

Fig. 6. Illustration for the former part of the proof of Lemma 7.

Prefix(t_ℓ), and thus we can detect them in $O(|label(s_\ell)| - |label(suf(s_{\ell-1}))| + 1)$ time and in $O(|label(t_\ell)| - |label(suf(t_{\ell-1}))| + 1)$ time, respectively, by using the suffix links. Then the total time cost for detecting s_ℓ and t_ℓ for all possible ℓ is proportional to

$$\sum_{\ell=j+1}^{i} \{(|label(s_\ell)| - |label(suf(s_{\ell-1}))| + 1) + (|label(t_\ell)| - |label(suf(t_{\ell-1}))| + 1)\}$$
$$= (|label(s_{j+1})| - |label(suf(s_j))| + 1) + (|label(t_{j+1})| - |label(suf(t_j))| + 1)$$
$$+ (|label(s_{j+2})| - |label(suf(s_{j+1}))| + 1) + (|label(t_{j+2})| - |label(suf(t_{j+1}))| + 1)$$
$$\cdots\cdots$$
$$+ (|label(s_i)| - |label(suf(s_{i-1}))| + 1) + (|label(t_i)| - |label(suf(t_{i-1}))| + 1)$$
$$= |label(s_i)| - |label(suf(s_j))| + |label(t_i)| - |label(suf(t_j))| + 4(i - j - 1) + 2$$
$$= |label(s_i)| - |label(s_j)| + |label(t_i)| - |label(t_j)| + 4(i - j)$$
$$= |\varepsilon| - |label(s_j)| + |x_k| - |label(t_j)| + 4(|x_k| - 1)$$
$$\leq |x_k| - |x_k| + 4(|x_k| - 1)$$
$$= 4(|x_k| - 1).$$

This operation for the detection is illustrated in Fig. 7. Of course, after each detection we create a new node v_ℓ for each s_ℓ, or possibly $v_\ell = s_\ell$, and reconnect to v_ℓ the out-going edge of t_ℓ leading to its certain child r_ℓ corresponding to string u_ℓ. This reconnection as well takes just constant time. □

Secondly, we consider the suffixes of w_k beginning at position h with $i \leq h \leq i + |x_k| - 1$. As seen in Fig. 5, the beginning positions of those suffixes become inactive after the substitution of A_k for x_k occurring at position i. It means that all of them have to be removed from the tree structure. Hereby we call the region from $i+1$ to $i + |x_k| - 1$ the *dead zone* for x_k with respect to position i. The suffixes in the dead zone are dark shaded in the right figure of Fig. 5.

Lemma 8. *At every k-th stage, it is feasible in $O(|x_k|)$ time to remove all paths spelling out a suffix of w_k which begins at a position in the dead zone of w_k.*

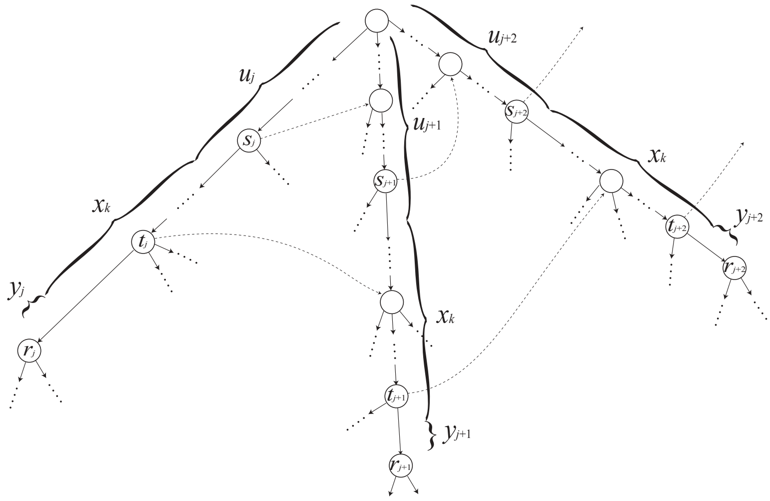

Fig. 7. Illustration for the latter part of the proof of Lemma 7.

Proof. Assume the path spelling out x_k is already converted to that spelling out a new character A_k. Remark there always exists a node v such that $label(v) = A_k$. Then, there exists $leaf_i$ in the subtree rooted at node v. It is trivial that $suf(leaf_i) = leaf_{i+1}$, and thus we can find it in constant time. By removing $leaf_{i+1}$ and its in-coming edge, we can delete the path spelling out the suffix $w_k[i+1:]$. Similarly it takes constant time for any h with $i+1 < h \leq i+|x_k|-1$. □

See Fig. 8 and Fig. 9 that show the trees after the first and second substitutions for the LRFs, respectively, with respect to string `abaaabbababb$`.

As stated above, we can maintain the data structure for $w_1, w_2, \ldots$. In this data structure, $BegPos_{w_k}(label(s))$ is exactly the set of leaves in the subtree rooted at node s. The sole remaining matter is, for each node s, to maintain the triple of the minimum element, the maximum element, and the cardinality of $BegPos_{w_k}(label(s))$. A short consideration reveals that we need the triples only for the nodes whose proper descendents represent non-repeating factors of w_k at the k-th stage. We can maintain the triples for such nodes only in linear time with respect to $|x_k| \cdot \#occ_{w_k}(x_k)$.

The last thing we have to clarify is how to deal with the node list from which we find the next LRF for substitution. One may think reordering the list is necessary after every substitution since some occurrences of the upcoming LRFs may disappear because of the previous LRF substitution. However, we in fact do not need to do that. If we encounter in the list a node that does not exist in the tree any more, then we just ignore it and focus on the next node in the list. Concerning the case that we encounter in the list a node s which still exists in the tree but $label(s)$ is *not* repeating any more, we do the followings. First,

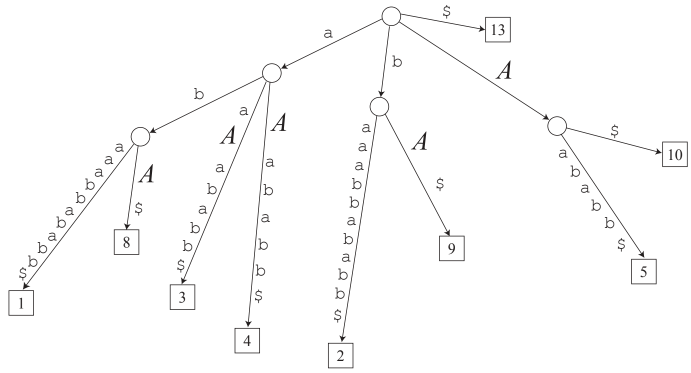

Fig. 8. The resulting tree structure for $\overline{w_2}$ = `abaa`A`ab`$A\$$. It is sufficient for us to find an LRF x_2. In fact, x_2 = `ab` is represented by an internal node.

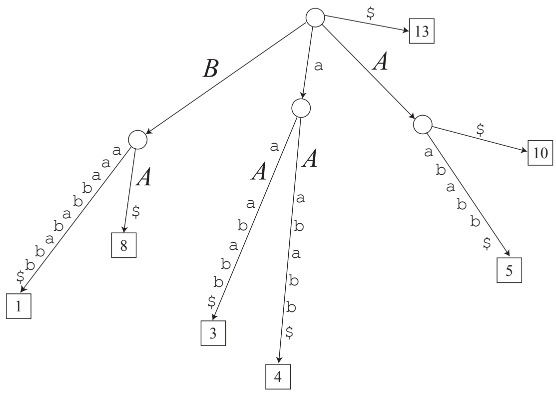

Fig. 9. The resulting tree structure for $\overline{w_3} = B$`aa`$ABA\$$. Since there is no internal node of length more than one, the encoding of the text halts here.

we focus on the subtree rooted at the node s and see its all leaf nodes. If the remainder of subtracting the maximum leaf number from the minimum one is less than $|label(s)|$, it implies that $label(s)$ is non-repeating. We then mark node s 'dead', and focus on the upcoming LRF in the list. If in traversing the subtree rooted at s we encounter any internal node t marked 'dead', then we do not traverse the subtree rooted at t. This way we can avoid touching the leaf nodes of the subtree for t more than once. The total time cost is therefore only linear in the number of the leaf nodes, which is $O(|w|)$.

Last, recall the proof of Lemma 7 where a possibility of creation of a new node v is mentioned. If $label(v)$ is a repeating factor of length more than one, then we insert v to the bin-sorted list for LRFs. This insertion can be done in constant time. The matter is how to examine if the new node v should be in the node list or not. The length check can be done in constant time by seeing *length*(v). Then we see all child nodes of v and their minimum and maximum beginning positions. Since the number of the child nodes of v is at most $|\Sigma|$, we can compute the minimum and maximum beginning positions i, j of v in constant time assuming Σ is fixed. If $j - i \geq \mathit{length}(v)$ then v is inserted into the list, and otherwise not. Clearly this calculation takes constant time.

We now have the main result of this paper.

Theorem 3. *The text compression based on the longest-first substitution is feasible in linear time.*

Proof. The preprocessing of input string w is feasible in $O(|w|)$ time. Let N be the number of stages in the compression of w. The k-th stage of the compression takes $O(|x_k| \cdot \#occ_{w_k}(x_k))$ time. Since $|x_k| \cdot \#occ_{w_k}(x_k) \leq 2(|x_k| - 1) \cdot \#occ_{w_k}(x_k) = 2(|\overline{w_k}| - |\overline{w_{k+1}}|)$, we obtain $\sum_{k=1}^{N} |x_k| \cdot \#occ_{w_k}(x_k) \leq 2\sum_{k=1}^{N}(|\overline{w_k}| - |\overline{w_{k+1}}|) \leq 2|\overline{w_1}| = O(|w|)$. □

4 Conclusions and Future Work

This paper introduced a linear-time algorithm to compress a given text by longest-first substitution. We employed a suffix tree in the core of the algorithm, gave some operations for updating the tree after the substitution for a longest repeating factor, and delved in the analysis of the accuracy and time complexity of the algorithm.

An interesting fact is that we can also use compact directed acyclic word graphs (CDAWGs) [6] that are smaller than suffix trees. Note that, though Proposition 4 relies on Ukkonen's suffix tree construction algorithm, the on-line algorithm of [10] could the same role for CDAWGs. However, the operation to maintain a CDAWG after the substitution for an LRF, is relatively more complicated, since it is a graph which has only one sink node. Namely, all suffixes of an input text are represented by one node, unlike the suffix tree with a one-to-one correspondence between a suffix and leaf node. However, it is possible in (amortized) constant time to simulate the suffix link traversal between two leaf nodes of a suffix tree in the corresponding CDAWG, by a technique similar to the one introduced in the latter part of the proof for Lemma 7.

The ultimate goal of off-line grammar-based text compression is to first replace the factor x of input string w with a new character, such that $\#occ_w(x) \times |x| \geq \#occ_w(y) \times |y|$ for any other $y \in \mathit{Factor}(w)$ [16]. Namely, the *largest-area-first* substitution mechanism. For this purpose, every node v of $\mathit{STree}(v)$ has to be annotated by $\#occ_w(\mathit{label}(v))$. It corresponds to the *minimal augmented suffix tree* (*MASTree*) of w [3, 2]. The size of $\mathit{MASTree}(w)$ is known to be $O(|w|)$, but there currently exists only an $O(|w| \log |w|)$-time algorithm for its construction [7]. Therefore, to achieve a linear-time algorithm for text compression by largest-area-first substitution, we first need to develop a linear-time construction algorithm for $\mathit{MASTree}(w)$. In addition, we need a linear-time solution for sorting nodes of the tree in the order of their 'areas', and it is also a challenging open problem.

References

[1] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer-Verlag, 1985.

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

[3] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15:481–494, 1996.

[4] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, New Jersey, 1990.

[5] 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.

[6] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

[7] G. S. Brødal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen. Solving the string stastistics problem in time $O(n \log n)$. In *Proc. 29th International Colloquium on Automata,Languages, and Programming (ICALP'02)*, volume 2380 of *LNCS*, pages 728–739. Springer-Verlag, 2002.

[8] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

[9] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[10] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *LNCS*, pages 169–180. Springer-Verlag, 2001.

[11] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 6th Annual International Conference on Computing and Combinatorics (COCOON'96)*, volume 1090 of *LNCS*, pages 219–230. Springer-Verlag, 1996.

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

[13] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[14] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artificial Intelligence Research*, 7:67–82, 1997.

[15] C. G. Nevill-Manning and I. H. Witten. Phrase hierarchy inference and compression in bounded space. In *Proc. Data Compression Conference '98 (DCC'98)*, pages 179–188. IEEE Computer Society, 1998.

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

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

[18] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[19] J. G. Wolff. An algorithm for the segmentation for an artificial language analogue. *Britich Journal of Psychology*, 66:79–90, 1975.

[20] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans Information Theory*, 24(5):530–536, 1978.