Technical Reports on Mathematical and Computing Sciences: TR-C190
title: Finding Maximal Repeats with Factor Oracles
author: Ryoichi KATO[1]
affiliation:
1. Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
Meguro-ku Ookayama, Tokyo 152-8552, Japan
(`Ryoichi.Kato@is.titech.ac.jp`)

Abstract: Factor oracles, built from an input text, are automata similar to suffix automata, and accepting at least all substrings of the input text. In papers [LL00] and [LLA02], factor oracles are used to detect repeats on text. Although repeats found with these methods are not maximal, average error is very low and algorithm runs quite fast.

In this paper, we present two ideas to improve accuracy of the algorithms in [LL00] and [LLA02]. One uses slightly modified algorithm of [LL00], but improves accuracy significantly, with slight overhead of running time. The other uses new property of factor oracles shown in [Kat03b] to detect maximal repeats of an input text (but slow).

1 Introduction

Factor oracles are deterministic acyclic automata built from a given text. Factor oracles are more space economical and easy to implement comparing to similar machineries such as suffix-trees [Wei73, McC76, Apo85, Ukk95] or suffix-automata [BBE$^+$85, CV97, IHS$^+$01]. It also runs with less memory than these machineries.

In [Kat03b] (English version: [Kat03a]), we revealed a new property of factor oracles, which leads us to a new full-text search algorithm using factor oracles as indexes. The property we found also related to problem of finding repeating substrings of a text using factor oracles, which is the main theme of this paper.

More precisely, the problem can be described as follows. For all prefixes $p[1 \dots i]$ ($1 \leq i \leq |p|$) of input text p, we want to compute length l of (one of / longest) repeated suffix $p[j \dots i]$ ($1 \leq j \leq i$) of the prefix, and its repeating position r ($r < i,\ p[r-l+1 \dots r] = p[j \dots i]$). Ideally, we want to compute the *longest* repeats over such repeated suffixes, for *all* prefixes of p, (hopefully) in linear time. Alternatively, finding fairly long repeat, instead of finding the longest ones, in exchange for running time would be a better solution for some applications. Finding such repeats lead us to applications such as analysis of genomic sequences [LLDA03] and data compression [LL01].

An algorithm to find such repeats is presented in [LL00] and improved version of the algorithm in [LLA02]. These algorithms can find fairly long repeats but not necessarily longest ones. In this paper, we present several improved algorithms based on these two algorithms and two new ideas. First, we present modified version of the algorithm in [LL00]. Although the modification is very simple and slight, it improves accuracy significantly, with slight over head of running time. Secondly, we present another algorithm to compute *longest* repeats, using first algorithm as well as a new property of factor oracles we have proved earlier ([Kat03a]). We also show combination of our ideas and [LLA02] leads to considerable improvement.

2 Basic Definitions and Properties of Factor Oracle

In this section, preliminarily notions, definitions and lemmas upon basic properties of factor oracles are introduced. Most of them appear in [ACR01].

First of all, let us define the notations and the definitions used throughout this document.

- Let $p = p[1 \dots m]$ be a string over an arbitrary alphabet Σ of length $|p| = m$. $(p \in \Sigma^*)$
- let ϵ be an empty string $|\epsilon| = 0)$
- Relation $a \overset{\text{suf}}{\prec} b$ ($\overset{\text{pre}}{\prec}$ respectively) denotes that a is a suffix (prefix) of b.
- Relation $a \overset{\text{fac}}{\prec} b$ denotes a is a substring of b.
- We denote by $\overline{w}$ the state in an automaton reading of string w ends.

Factor oracle is introduced in [ACR01] along with related concepts to it on strings such as *longest repeated suffix* ($\mathit{repet}_p(i)$), *minimal length path* ($\mathit{min}(i)$), etc.

Definition 2.1 ([ACR01]). $\mathit{Oracle}(p)$, defined by its construction algorithm BUILD_ORACLE in [ACR01], is an acyclic automaton which recognizes at least all of substrings of p. It has exactly $m + 1$ states(nodes), exactly m internal transitions, and at most $m - 1$ external transitions. Note that all of states are terminating. It also has exactly m additional links called *suffix links* which are not part of automaton.

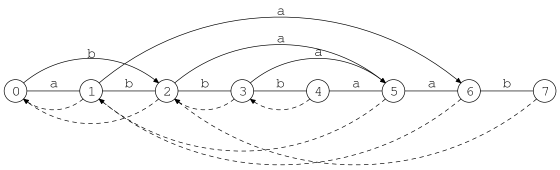

Figure 1: Oracle("abbbaab")
Internal transitions are shown as straight line. Solid arcs left to right on upper side are external transitions. Dashed arcs right to left on downside are suffix links.
Note that the internal transitions has one-to-one correspondence with letters of the input text.

Figure 1 illustrates Oracle("abbbaab")

- $\mathit{endpos}_p(w) = \{i | p = uwp[i] \dots p[m]$ for appropriate $u \in \Sigma *\}$
  Set of all ending position of occurrences of w in p.
- $\mathit{poccur}(w, p) = \min\{\mathit{endpos}_p(w)\}$ (Ending position of the first occurrence of w in p)
- Let $\mathit{min}(i)$ be the minimal length string recognized at state i.
- Let $\mathit{repet}_p(i)$ be the longest suffix of prefix $p[0 \dots i]$ of p, which occurs at least twice in $p[0 \dots i]$. *Longest repeated suffix of* $p[1 \dots i]$

Suffix links are merely an aid of online construction and are not components of factor oracle if we interpret factor oracle as plain automaton. Nevertheless, suffix links play very important role in our algorithm. Suffix links are shown as dotted arc in figure 1.

Definition 2.2 ([ACR01]). $S(i)$ of state $i \in \mathit{Oracle}(p)$ is equivalent to $\overline{\mathit{repet}_p(i)}$. We arbitrary set $S(0) = -1$

Definition 2.3 ([ACR01]). Suffix path $\mathit{SP}(i)$ is defined as a set of states on the suffix path starts at state i. i.e., $\mathit{SP}(i) = \{i, \mathit{S}(i), \mathit{S}(\mathit{S}(i)), \ldots, 0\}$

Definition 2.4. Let $\mathit{Bros}(i)$, brothers of state $i \in \mathit{Oracle}(p)$ be a set $\{j | j \in \mathit{S}^{-1}(\mathit{S}(i))\}$.

Likewise, let $\mathit{LBros}(i)$, left brothers of i be a set $\{j | j \in \mathit{S}^{-1}(\mathit{S}(i))$ and $j < i\}$

[ACR01] shows many interesting, and important lemmas on factor oracle. Among them, we use the following ones.

Lemma ([ACR01] Lemma 1). Let $u \in \Sigma^*$ be a minimal length string among the strings recognized at state i (i.e., $\mathit{min}(i) = u$) of $\mathit{Oracle}(p)$. Then $u \overset{\text{fac}}{\prec} p$ and $i = \mathit{poccur}(u, p)$

Lemma ([ACR01] Lemma 2). Let i be a state of $\mathit{Oracle}(p)$ and $u = \mathit{min}(i)$. Then u is a suffix of any string $v \in \Sigma^*$ recognized at state i

Lemma ([ACR01] Lemma 3). Let $w \overset{\text{fac}}{\prec} p$. w is recognized by $\mathit{Oracle}(p)$ in a state $j \leq \mathit{poccur}(w, p)$

Lemma ([ACR01] Lemma 4). Let i be a state of $\mathit{Oracle}(p)$ and $u = \mathit{min}(i)$. Any path ending by u leads to a state $j \geq i$

Lemma ([ACR01] Lemma 7). Let $s > 0$ be a state of $\mathit{Oracle}(p)$ such that $\mathit{S}(s)$ is not 0. Then $\mathit{repet}_p(\mathit{S}(s)) \overset{\text{suf}}{\prec} \mathit{repet}_p(s)$.

Finally, we introduce the property of factor oracle we presented in [Kat03b] concisely, along with related definitions.

Definition 2.5 ([Kat03b]). $\mathit{S}^{-1}(i)$ is a set of all states j of $\mathit{Oracle}(p)$ such that $\mathit{S}(j) = i$ [1]

Corollary 2.1. Since each state $i \in \mathit{Oracle}(p)$ has exactly one suffix link destination $\mathit{S}(i) < i$, entire suffix links of a factor oracle can be interpreted as a tree structure. The state 0 is root of such tree and for any node(state) i, its parent is equal to $\mathit{S}(i)$, and children are equal to $\mathit{S}^{-1}(i)$ which may be $\emptyset$. We call such a tree *suffix-link-tree*.

Definition 2.6 ([Kat03b]). $\mathit{SP}^{-1}(i)$ is a set of all of descendant nodes of i in the suffix-link-tree of $\mathit{Oracle}(p)$ i.e., $\mathit{SP}^{-1}(i) = \{j | i \in \mathit{SP}(j)\}$

In other words, $\mathit{SP}^{-1}(i)$ is set of all states in a subtree of suffix-link-tree where i is the root of such subtree.

Theorem ([Kat03b]). Set of all ending position of w's occurrence in p, $\mathit{endpos}_p(w)$ can be described as follows.

$$\mathit{endpos}_p(w) = \begin{cases} \{\overline{w}\} \cup \{k \in \mathit{SP}^{-1}(j) \,\big|\, \text{for all } j \in \mathit{S}^{-1}(\overline{w}) \text{ s.t. } w \text{ appears at } j\ \} \\ \qquad\qquad \text{if } \overline{w} = \mathit{poccur}(w, p) \\ \\ \{k \in \mathit{SP}^{-1}(j) \,\big|\, \text{for all } j \in \mathit{S}^{-1}(\overline{w}) \text{ s.t. } w \text{ appears at } j\ \} \\ \qquad\qquad \text{otherwise} \end{cases}$$

[1] Intuitively, $\mathit{S}^{-1}(i)$ means tracing all of incoming suffix link in inverse direction

Corollary 2.2. When string w is accepted by $\mathsf{Oracle}(p)$, following holds.

$$w \overset{\text{fac}}{\prec} p \iff w \text{ appears at at least one state } j \text{ such that } j \in \{\overline{w}\} \cup \mathsf{S}^{-1}(\overline{w})$$

These properties of factor oracle leads us to the following important corollary.

Corollary 2.3. $\mathsf{repet}_p(i)$ appears at at least one state j such that $j \in \mathsf{LBros}(i) \cup \{\mathsf{S}(i)\}$.

Proof. Definition assures $\mathsf{repet}_p(i) \overset{\text{fac}}{\prec} p$. Using corollary 2.2 and the fact that reading of $\mathsf{repet}_p(i)$ ends in $\mathsf{S}(i)$ by the definition of factor oracle. $\square$

3 Finding Repeats Using Factor Oracle

In [LL00], an heuristic algorithm to find repeats using factor oracle is presented. The algorithm computes, for all prefixes $p[1 \dots i]$ of p, length of one of its repeated suffix. Although computed repeats are not necessarily longest, experimental result shows that algorithm outputs a good approximation of longest repeats. Further more, an improved version of this algorithm is presented in [LLA02], which significantly lowers error and error rate.

In this section, we present new algorithms based on above two algorithms. First algorithm is a new improved algorithm of [LL00]. Although accuracy of outputs are about same as of another improvement in [LLA02], our algorithm is better with regarding running time and space efficiency. The other algorithm we present computes longest repeated suffixes precisely, using corollary 2.3 (but slower).

Since improvement in [LLA02] does not conflict with ours, we can combine these improvements together to obtain further-more-improved algorithms. However, combination makes analysis even more difficult and exact characteristics of these algorithms are not yet known. So we only present experimental results, which are astonishingly improved with relatively small overhead, in the next section.

3.1 Algorithm of [LL00] (lrs: length repeated suffix)

In [LL00], it is proved that $\mathsf{lrs}_p(i)$, inductively defined as definition 3.2, gives length of one of repeated suffix of $p[1 \dots i]$, and the repeated suffix appears at $\mathsf{S}(i)$ (i.e., $p[\mathsf{S}(i) - \mathsf{lrs}_p(i) + 1 \dots \mathsf{S}(i)] \overset{\text{suf}}{\preceq} \mathsf{repet}_p(i)$). They also proved that the algorithm can be ran simultaneously with construction of factor oracle, and total running time of the algorithm is linear to the length of input text $|p|$.

Definition 3.1 ([LL00]). Let j be a minimal states among all state those has transition to state $i+1 \in \mathsf{Oracle}(p)$. Then, $\pi_1(i+1)$ is equal j. $\pi_2(i+1)$ is equal to the state $k \in \mathsf{SP}(\mathsf{S}(i+1)-1)$ such that $\mathsf{S}(k) = \mathsf{S}(j)$ if $\mathsf{S}(i+1) \neq \mathsf{S}(k)$ or is equal to $\mathsf{S}(j)$ if $\mathsf{S}(i+1) = \mathsf{S}(k)$. We simply denote $\pi_1(i+1), \pi_2(i+1)$ as π_1, π_2 when there is no concern of confusion.

Definition 3.2 ([LL00]).

$$\mathsf{lrs}_p(i+1) = \begin{cases} 0 & \text{if } \mathsf{S}(i+1) = 0 \quad (1) \\ \mathsf{lrs}_p(\pi_1) + 1 & \text{if } \pi_2 = \mathsf{S}(\pi_1) \quad (2) \\ \min(\mathsf{lrs}_p(\pi_1), \mathsf{lrs}_p(\pi_2)) + 1 & \text{otherwise.} \quad (3) \end{cases}$$

We arbitrary set $\mathsf{lrs}_p(0)$ to 0.

3.2 Longest Common Suffix of i and $\mathsf{S}(i)$

In this subsection, we introduce the definition of $\mathit{lcs}_p(i)$, which is a base of our new improved algorithm and prove that $\mathit{lcs}_p(i)$ always gives the longest common suffix of i and $S(i)$.

Lemma ([Kat03b] Lemma 2.2). For a state $i \in \mathit{Oracle}(p)$ and for a state j such that $j = S(i)$, $\mathit{min}(j) \overset{\text{suf}}{\prec} \mathit{min}(i)$[2].

Corollary 3.1. For a state $i \in \mathit{Oracle}(p)$ and for a state j such that $j = S(i)$, $\mathit{repet}_p(j) \overset{\text{suf}}{\precneqq} \mathit{min}(j) \overset{\text{suf}}{\preceq} \mathit{repet}_p(i) \overset{\text{suf}}{\precneqq} \mathit{min}(i)$

Proof. By [Kat03b]—Lemma 2.2 and [ACR01]—Lemma 1. □

Definition 3.3.

$$\mathit{lcs}_p(i+1) = \begin{cases} 0 & \text{if } S(i+1) = 0 & (1) \\ \mathit{lcs}_p(\pi_1) + 1 & \text{if } \pi_2 = S(\pi_1) & (2) \\ \text{compute the length directly.} & \text{if } \mathit{lcs}_p(\pi_1) = \mathit{lcs}_p(\pi_2) & (3') \\ \min(\mathit{lcs}_p(\pi_1), \mathit{lcs}_p(\pi_2)) + 1 & \text{otherwise.} & (3) \end{cases}$$

We arbitrary set $\mathit{lcs}_p(0)$ to 0

Lemma 3.1. $\mathit{lcs}_p(i)$ is equal to the longest common suffix of i and $S(i)$.

Proof. By induction. Notice that length of a common suffix of $p[1 \dots i+1]$ and $p[1 \dots S(i+1)]$ is equal to a common suffix of $p[i]$ and $p[1 \dots S(i+1)-1]$, plus 1, as remarked in [LL00].

$i = 1$: $S(i) = 0$ and $\mathit{lcs}_p(i) = 0$ condition (1) is verified.

$i > 0$: Assume that $\forall j,\ 1 \le j \le i.$ then $\mathit{lcs}_p(j)$ is equal to the longest common suffix of $S(j)$ and j. Notice that, under this assumption, following statement always holds.

- For arbitrary two states $k < l$ such that $k \in SP(l)$, longest common suffix of $p[1 \dots k]$ and $p[1 \dots l]$ is equal to, for a state m such that $S(m) = k$, longest common suffix of $p[1 \dots k]$ and $p[1 \dots m]$ which is equal to $\mathit{lcs}_p(m)$. — $\star$

Then, as defined, four cases have to be considered for computation of $\mathit{lcs}_p(i+1)$.

- case (1): $S(i) = 0$ verifies $\mathit{lcs}_p(i)$, length of the common suffix of ϵ and $p[1 \dots i]$, is equal to 0 regardless of i.
- case (2): By $\star$, the length of the longest common suffix of $p[1 \dots i]$ and $p[1 \dots S(i+1)-1]$ is equal to $\mathit{lcs}_p(\pi_2)$.
- case (3'): Trivial.
- case (3): In this case, we can assume $\mathit{lcs}_p(\pi_1) < \mathit{lcs}_p(\pi_2)$ without losing generality.

  Since $\mathit{lcs}_p(\pi_1)$ is equal to length of the longest common suffix of $p[1 \dots j]$ and $p[1 \dots pi_1]$ by the induction hypothesis, $p[\pi_1 - \mathit{lcs}_p(pi_1)] \neq p[j - \mathit{lcs}_p(pi_1)]$. Then, by the assumption of $\mathit{lcs}_p(\pi_1) < \mathit{lcs}_p(\pi_2)$, $p[\pi_1 - \mathit{lcs}_p(\pi_1)] \neq p[\pi_2 - \mathit{lcs}_p(\pi_1)]$ because

[2] This means that $\mathit{min}(j)$ is one of common suffix of $p[1 \dots j]$ and $p[1 \dots i]$

$p[\pi_2 - \mathsf{lcs}_p(\pi_1)] = p[j - \mathsf{lcs}_p(\pi_1)]$. Hence length of longest common suffix of $p[1 \ldots \pi_1]$ and $p[1 \ldots \pi_2]$ is equal to $\mathsf{lcs}_p(\pi_1)$ Beside, we know, for arbitrary two decendent nodes of $m \in \mathsf{SP}^{-1}(\pi_1)$ and $n \in \mathsf{SP}^{-1}(\pi_2)$, $p[m - \mathsf{lcs}_p(\pi_1)] \neq p[n - \mathsf{lcs}_p(\pi_1)]$ from corollary 3.1. So, longest common suffix of any two nodes in $\mathsf{SP}^{-1}(\pi_1)$ and $\mathsf{SP}^{-1}(\pi_2)$ including i and $\mathsf{S}(i + 11) - 1$ is equal to that of $p[1 \ldots \pi_1]$ and $p[1 \ldots \pi_2]$, which in tern, is equal to $\mathsf{lcs}_p(\pi_1)$.

proof of $\star$: Let m states on the suffix path $\mathsf{SP}(l)$ from l to $k \in \mathsf{SP}(l)$ be $\{k = j_1, j_2, j_3, \ldots, j_{m-1}, j_m = l\}$, $j_n = \mathsf{S}(j_n + 1)$. Then, by induction hypothesis, $\forall n$, $1 \leq n \leq m$, $p[j_n - \mathsf{lcs}_p(j_{n+1})] \neq p[j_{n+1} - \mathsf{lcs}_p(j_{n+1})]$. Besides, corollary 3.1 assures $|\mathsf{lrs}_p(j_n)| < |\mathsf{lrs}_p(j_{n+1})|$.

Then $\forall n$, $2 \leq n \leq m$, $p[j_1 - \mathsf{lcs}_p(j_2)] \neq p[j_{n+1} - \mathsf{lcs}_p(j_2)]$, which mean the length of the longest common suffix of $p[1 \ldots k = j_1]$ and $p[1 \ldots j_n]$ is equal to the length of the longest common suffix of $p[1 \ldots k]$ and $p[1 \ldots j_2]$, and this is equal to $\mathsf{lcs}_p(j_2)$. □

Although the definition of $\mathsf{lcs}_p(i)$ we have presented has only a very simple difference with that of $\mathsf{lrs}_p(i)$ — which only introduces ($3'$) to the definition of $\mathsf{lrs}_p(i)$ — difference between computed values are significant. $\mathsf{lcs}_p(i)$ always computes the longest common suffix of $p[1 \ldots i]$ and $p[1 \ldots \mathsf{S}(i)]$ buf $\mathsf{lrs}_p(i)$ does not.

In case ($3'$), we have to compute the length directly by comparing a letter by a letter beginning from i and $\mathsf{S}(i)$, in decreasing order, until the smallest value l such that $p[i-l] \neq p[\mathsf{S}(i) - l]$ is found. Then l is the length of the longest common suffix of $p[1 \ldots i]$ and $p[1 \ldots \mathsf{S}(i)]$. But we know that, even in case ($3'$), $l' = \min\{\mathsf{lcs}_p(\pi_1), \mathsf{lcs}_p(\pi_2)\}$ is the length of one of common suffix of $p[1 \ldots i]$ and $p[1 \ldots \mathsf{S}(i)]$ and this value approximates the length of the longest common suffix well. So, we can start comparison from $i - l'$ and $\mathsf{S}(i) - l'$ instead of i and $\mathsf{S}(i)$.

Then overhead of running time is linear to the sum of improvements $\mathsf{lcs}_p(i) - l'$ for all i such that ($3'$) is applied. This overhead is experimentally very small and our algorithm runs faster than that of [LLA02], another improved version of $\mathsf{lrs}_p(i)$, as mentioned in the next section. Note that overall improvement in computed length is greater than the overhead (= sum of individual improvements), because the definition is inductive and a computed value may be referred afterward.

Furthermore, regarding space efficiency, our algorithm is better than that of [LLA02], which needs inverse suffix links, because it requires extra memory space about 12 bytes per a byte of input to implement. (Assuming a simple linked-list-based implementation with 4byte integer).

3.3 Finding the Longest Repeated Suffix (LRS)

According to corollary 2.2, the longest repeated suffix of $p[1 \ldots i]$, $\mathsf{repet}_p(i)$ is equal to $\max\{l | l = \mathsf{lrs}_p(j) s.t. j \in \mathsf{LBros}(\mathsf{S}(i))\}$. Or, in the midst of construction, it is equal to $\max\{l | l = \mathsf{lrs}_p(j)\text{s.t.} j \in \mathsf{Bros}(\mathsf{S}(i))\}$ for $\mathsf{Oracle}(p[1 \ldots i])$.

In order to search all (left) brothers $\mathsf{Bros}(\mathsf{S}(i))$, it will take $\mathcal{O}(\text{repeat length} \times |\mathsf{Bros}(\mathsf{S}(i))|)$ of time for each i, it can be very worse especially for inputs with long repeats in it. So we considered using $\mathsf{lcs}_p(i)$. Note that we only have to look $j \in \mathsf{Bros}(i)$ such that $\mathsf{lrs}_p(i) = \mathsf{lrs}_p(j)$ among brothers. This can be easily proved with an argument similar to the proof of $\star$ in lemma 3.1. Then we can start comparison from $j - \mathsf{lrs}_p(i)$ and $i - \mathsf{lrs}_p(i)$ instead of j and i because we know $p[j - \mathsf{lrs}_p(i) + 1 \ldots j] = p[i - \mathsf{lrs}_p(i) + 1 \ldots i]$.

With the following algorithm, the longest repeated suffix $\mathit{LRS}_p(i) = \mathit{repet}_p(i)$, for precalculated $\mathit{Oracle}(p[1 \dots i])$ and $\mathit{lcs}_p(1) \sim \mathit{lcs}_p(i)$, is equal to FindBestAmongBros ($\mathit{Oracle}(p[1 \dots i])$, $\mathit{lcs}_p(i)$) Our algorithm also can be run simultaneously with construction of a factor oracle.

Algorithm 1 FindBestAmongBros($\mathit{Oracle}(p[1 \dots i])$, $\mathit{lcs}_p(i)$)

$k := 0$; $p := 0$
<u>foreach</u> $j \in \mathit{LBros}(i)$ of $\mathit{Oracle}(p[1 \dots i])$
  <u>if</u> $\mathit{lcs}_p(j) = \mathit{lcs}_p(i)$
    $k' :=$ length of longest common suffix of $p[1 \dots j - \mathit{lcs}_p(j)]$ and $p[1 \dots i - \mathit{lcs}_p(i)]$
    $p' := j$
    <u>if</u> $k' > k$
      $k = k'$; $p = p'$
<u>return</u> length $\mathit{lcs}_p(i) + k$ and position p

This strategy is somewhat similar to that of [LLA02], but differs in two points. First, our algorithm is greedy, try to find the longest improvement among brothers. On the other hand, algorithm of [LLA02] stops searching only for one letter of improvement in length. Secondly, algorithm of [LLA02] updates suffix links dynamically to a state where a improved length found. So, attained structure, called repeat oracle, is different from factor oracle.

4 Experimental Results

	induction	$\mathit{Bros}(i)$ search	$S(i)$ update	$S^{-1}(i)$	note
lrs	lrs	none	–	disused	[LL00]
lcs	lcs	none	–	disused	computes longest common suffix of $p[1 \dots i]$ and $p[1 \dots S(i)]$
ilrs	lrs	non-greedy	yes	required	[LLA02]
ilcs	lcs	non-greedy	yes	required	[LLA02] + lcs
glcs	lcs	greedy	yes	required	greedy version of ilcs
LRS	lcs	greedy	no	required	computes $\mathit{repet}_p(i)$ (corollary2.2)

Table 1: List of algorithms

lrs is induction of definition 3.2 ([LL00]). lcs is by definition 3.3. / Non-greedy brother search (with suffix link update) is originated in [LLA02]. Greedy brother search uses FindBestAmongBros in section 3.3. / Algorithm of [LL00] or [LLA02] are shown as *italics*. Our algorithms are shown as **bold face**.

We give here some experimental result with all 1024KB data and program implemented by programing language C to evaluate performance of algorithms we have presented. As mentioned in the previous section, we can combine several improvements together to obtain further-more-improved algorithms. So we also performed experiments on two different combination of algorithms named ilcs and glcs along with our algorithms lcs, LRS, the original algorithm lrs in [LL00], and its improved version ilrs in [LLA02]. ilcs is a combination of ilrs and lcs and glcs is a greedy-search version of ilcs. Table 1 shows combinations of these algorithms. Note that, using $S^{-1}(i)$ (inverse suffix links) require extra memory space about 12 bytes per a byte of input. (Assuming a simple linked-list-based implementation with 4byte integer). Besides, note that updating suffix links change structure of automaton and obtained automaton is somewhat different from factor oracle. In the case of ilrs, such a new structure is called *repeat oracle*([LLA02]) The characteristics of these new structures are not yet fully understood.

Table 2 shows running time of algorithms. There is only small amount of slowdown in lcs compared to lrs, and in ilcs compared to ilrs. Although both glcs and LRS performs greedy brother search, running speed of glcs is even comparable to ilcs (non-greedy search) while LRS runs very slowly on several inputs.

data	**LRS**	*lrs*	**lcs**	*ilrs*	***ilcs***	***glcs***
DNA-1	694	157	165	246	267	303
DNA-2	693	159	164	268	273	311
PostScript-1	225	129	132	182	190	202
PostScript-2	203	136	137	179	181	183
TrueTypeFont	259	193	193	230	240	241
bible-txt	219	117	117	173	168	174
photo-bitmap	463	380	382	430	442	444
random	464	412	437	456	463	459
world192-txt	190	116	119	163	164	172

Table 2: running time (1/100 sec.)

Table 3 shows error rate and average error of algorithms. Term error rate here is, for all i $(1 \leq i \leq |p|)$, rate of i such that the value $\mathit{lrs}_p(i)$ ($\mathit{lcs}_p(i)$, $\mathit{ilrs}_p(i)$, $\mathit{ilcs}_p(i)$, $\mathit{glcs}_p(i)$, resp.) differs from $\mathit{LRS}_p(i)$. Likewise, average errors are differences between average length of longest repeated suffixes: $\left(\sum_{i=1}^{|p|} |\mathit{repet}_p(i)|\right)/|p| = \left(\sum_{i=1}^{|p|} \mathit{LRS}_p(i)\right)/|p|$ and average of computed lengths of respective method. According to the result shown in the table, our lcs is comparable with ilrs regarding both error rate and average error. The result shows that ilcs gives a certain amount of improvement to ilrs. However glcs makes much more improvement. For most of data on which we experimented, excepting DNAs, glcs gives error rate of less than 0.5% and average error of less than 0.05 regardless of average length of longest repeated suffix.

	error rate (%)					average errors of computed length					
	lrs	**lcs**	*ilrs*	**ilcs**	**glcs**	(LRS)	*lrs*	**lcs**	*ilrs*	**ilcs**	**glcs**
DNA-1	60.94	45.60	39.10	32.76	2.87	12.94	1.10	0.85	0.68	0.59	0.04
DNA-2	59.26	43.53	37.62	31.41	2.83	9.44	1.05	0.80	0.63	0.54	0.04
PostScript-1	22.16	11.17	14.11	9.63	0.28	21.93	0.46	0.19	0.25	0.17	0.00
PostScript-2	19.29	8.72	16.21	8.92	0.15	205.72	2.60	0.67	2.58	0.94	0.00
TrueTypeFont	19.67	5.16	9.30	5.91	0.14	1994.77	0.98	0.34	0.62	0.42	0.01
bible-txt	35.17	15.16	23.32	15.16	0.52	15.42	1.00	0.40	0.61	0.42	0.01
photo-bitmap	4.02	2.87	2.61	2.18	0.07	3.36	0.05	0.04	0.03	0.03	0.00
random	0.01	0.01	0.01	0.01	0.00	1.97	0.00	0.00	0.00	0.00	0.00
world192-txt	20.46	9.68	13.11	8.97	0.21	20.18	0.96	0.43	0.74	0.51	0.01

Table 3: error rate and average error
column of LRS shows average length of longest repeated suffixes, rest of columns shows average difference.

Table 4 shows that case (3′) of definition3.3 are applied for 30% to 98% of suffixes. Among them, lengths are actually improved only on at most a tenth of suffixes. The average of individual improvement is 0.4 of length. Since overhead of running time is pro-

	A: case (3′) (%)			B: avg. improve.			C: imprv.occ.(%)		
	lcs	ilcs	glcs	lcs	ilcs	glcs	lcs	ilcs	glcs
DNA-1	32.01	28.46	22.04	0.38	0.21	0.04	9.96	5.01	0.86
DNA-2	31.33	28.14	22.36	0.39	0.21	0.04	9.84	4.72	0.85
PostScript-1	27.67	27.37	26.39	0.15	0.10	0.00	2.15	1.50	0.06
PostScript-2	26.80	26.60	26.14	0.07	0.04	0.00	0.79	0.44	0.02
TrueTypeFont	40.77	40.54	40.26	0.08	0.04	0.00	0.71	0.42	0.03
bible-txt	16.15	15.24	14.19	0.40	0.23	0.02	2.71	1.57	0.14
photo-bitmap	71.67	71.50	71.24	0.01	0.00	0.00	0.51	0.26	0.01
random	98.43	98.43	98.43	0.00	0.00	0.00	0.00	0.00	0.00
world192-txt	17.45	16.98	16.42	0.22	0.12	0.02	1.33	0.69	0.06

Table 4: Improvement by (3′) of definition3.3

A: Percentage of suffixes on which case (3′) is applied. (including suffixes with no improvements.) B: Average improvement in length with case (3′). C: Percentage of suffixes whose lengths are improved at least by 1 with case (3′).

portional to improved length, this slight improvement makes very small effect to running time of the algorithms.

However, as we noted in the previous section, overall improvement is greater than sum of the individual improvements. For example, experiments on lcs on PostScript-1 shows that improvement happens only for 0.79% of its suffixes and average of individual improvement in length is equal to 0.07, as shown in table 4. On the other hand, according to table 3, total improvement of lcs to lrs is 10.57% of drop in error rate and average error is lowered to 0.67 from 2.60.

5 Conclusion and Open Problems

We have presented two ideas to improve length of repeats found with factor oracles, based on method of [LL00]. First, we have presented lcs (Longest Common Suffix), an improved algorithm of lrs in [LL00]. While it runs with only a slight overhead and its running time is comparable to lrs, its accuracy is rather comparable to that of another improved method in [LLA02] (in this paper, we have called it ilrs).

Secondly, we have presented LRS, an algorithm to compute longest repeated suffixes. Although this method is rather slow, new property of factor oracle proved in [Kat03b] guarantees that computed lengths are maximal. This method also uses lcs for time efficiency.

In addition to above two algorithms, We have examined combinations of these ideas of improvement, since these ideas do not conflict each other. Among them, a method named glcs(Greedy-lcs), combination of lcs and ilrs with greedy search, performs very well both in running time and accuracy on experiment.

However, it seems to be very difficult to characterize exact approximation rate or running time of these algorithms theoretically. Even there remain open problems in fundamental characterization of factor oracles such as exact characterization of language recognized by factor oracles. Moreover, dynamic updated of suffix links, which is the key idea of ilrs, makes analysis much more difficult. So, we gave experimental result to show efficiency of these algorithms.

As future works, we want to analysis above-mentioned open problems. Especially, we are interested in characterizing factor of input data which affects performance of the algorithms.

References

[ACR01] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient experimental string matching by weak factor recognition. *Lecture Notes in Computer Science*, 2089:51–??, 2001.

[Apo85] A. Apostolico. The myriad virtues of suffix trees. In *In Combinatorial Algorithms on Words, NATO ISI Series*, pages 85–96. Springer-Verlag, 1985.

[BBE+85] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen, and J. Selferas. The smallest automaton recognizing the subwords of a text. In *Theoretical Computer Science*, volume 40, pages 31–55, 1985.

[CV97] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, 1264*, pages 116–129, Aarhus, Denmark, 1997. Springer-Verlag, Berlin.

[IHS+01] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pages 169–180, 2001.

[Kat03a] Ryoichi Kato. A new full-text search algorithm using factor oracle as index(in japanese). Technical Report C-185, Dept. of Math. and Comp. Sciences Research Reports, Tokyo Institute of Technology, December 2003.

[Kat03b] Ryoichi Kato. A new text search algorithm using factor oracle as full-text index. Technical Report COMP2003-50(2003-10) Vol.103 No.394 ISSN 0919-6072, Technical Report of IEICE, October 2003.

[LL00] A. Lefebvre and T. Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australia, 2000.

[LL01] A. Lefebvre and T. Lecroq. Compror: on-line lossless data compression with a factor oracle. In *Information Processing Letters*, volume 83(1), pages 1–6, 2001.

[LLA02] A. Lefebvre, T. Lecroq, and J. Alexandre. Drastic improvements over repeats found with a factor oracle. In *Proceedings of the 13th Australasian Workshop on Combinatorial Algorithms (AWOCA 2002)*, pages 253–265., 2002.

[LLDA03] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. Forrepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 3(14):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 1–11, 1973.