ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

A detail analysis on factor oracle construction of computing repeated factors

Hisashi Iwasaki

January 2006, C–219

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

A detail analysis on factor oracle construction of computing repeated factors

Hisashi Iwasaki
C/O Prof. Osamu Watanabe,
Dept. of Mathematical and Computing science,
Tokyo Institute of Technology
email: iwasaki2@is.titech.ac.jp

Research Report C−219

abstract

We show a detail implementation for a linear time and space method, introduced in [3], to compute the length of a repeated suffix for each prefix of a given word p. This method is based on the utilization of the factor oracle [1] of p, which is deterministic acyclic automata accepting all subustrings of p.

keyword: factor oracle, suffix link, repetition

1 Introduction

There exist many studies of finding repetitions in a given word p in areas such as bioinformatics and data compression. In [3] an on-line heuristic method, linear time and space in $|p|$, to compute, for each prefix $p[1 \dots i]$ of p, the length of one of its repeated suffix was proposed. This length is denoted by $lrs[i]$. This method, based on factor oracle [1], gives an efficient way for searching repetitions, which has been shown quite useful in practical applications, e.g. repetition search in genomic sequences. Furthermore in [4] on-line data compression scheme using this method was proposed.

Unfortunately, however, the worst-case complexity of constructing a factor oracle while computing lrs is not clear from the description of the method stated in [3]. We show here a detail implementation for a linear time and space method to compute the length of a repeated suffix for each prefix of a given word p. From our implementation, it is now clear that a factor oracle and the table for its lrs function can be constructed within linear time and space in $|p|$.

2 Notations

We will use standard notions and notations on strings such as $|p|$, the length of a string p, etc. Let Σ be our alphabet, we assume that all strings $p = p_1 p_2 \dots p_m$ are strings over Σ. A *factor* or *substring* (resp. *prefix suffix*) of p is a string w (resp. u, v) such that $p = uwv$ for some $u, v \in \Sigma^*$; in particular, for an i and j, $1 \leq i \leq j \leq m$, we use $p[i \dots j]$ to denote the substrings of p appearing from the ith character to the jth character.

For a given string p, a factor oracle $\mathit{Oracle}(p)$ is an automaton with the following features:
• it is an acylic,
• it consists of $|p|+1$ states (which are all accepting states) and $|p|$ to $2|p|-1$ transitions, and
• it accepts all factors of p.

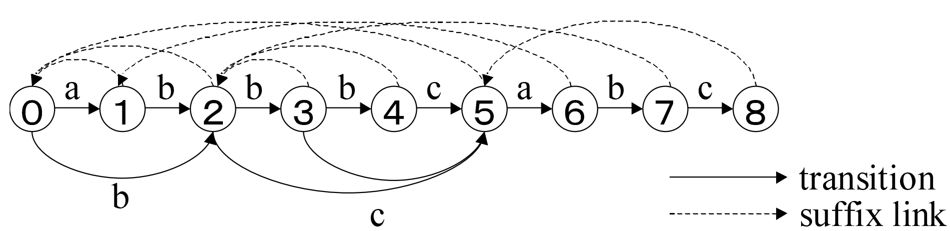

Figure. 1: $Oracle(abbbcabc)$

For example, a factor oracle $\mathit{Oracle}(p)$ for $p = abbbcabc$ is given as Figure 1. Here state 0 is the initial state. On this figure, the reader can check that it accepts all substrings of p; it is also easy to check that this factor oracle accepts "abc", nonsubstring of p; that is , it makes an error acceptance.

Definition 1. $\text{repet}_p(i)$ is the longest suffix of $p[1 \dots i]$ that appears at least twice in $p[1 \dots i]$.

For example, in Figure 1, $\text{repet}_p(1) = \epsilon$, $\text{repet}_p(4) = bb$, $\text{repet}_p(8) = abc$.

Definition 2. A function S_p maps each state $i > 0$ of $\mathit{Oracle}(p)$ to state j in which the reading of $\text{repet}_p(i)$ ends($S_p(i) = j$). For completeness, we set $S_p(0) = -1$. We call $S_p(i)$ suffix link of the state i in $\mathit{Oracle}(p)$.

Definition 3. We denote $k_0 = i, k_j = S_p(k_{j-1})(j \geq 1)$ for any state $i > 0$. The sequence of the k_i is finite, strictly decreasing and ends in state 0.

We call this sequence of states a suffix path, and define $SP_p(i)$ to be the set of states on the suffix path from i, that is, $SP_p(i) = \{k_0 = i, k_1 = S_p(i), \dots, k_t = 0\}$.

3 Computing repeated suffix with factor oracle

In [3] an on-line heuristic algorithm to compute, for each prefix $p[1 \dots i]$, the length of one of its repeated suffixes, such that $S(i)$ is one of its occurrences. This length is denoted by $lrs[i]$. In this section we propose another proof about the time complexity of this algorithm.

First we explain briefly about definition of lrs. During the construction of $Oracle(p[1 \dots i+1])$ from $Oracle(p[1 \dots i])$ and p_{i+1}, the backward jumps on the suffix path $SP_p(i)$ ends when a state j is reached such that $\delta(j, p_{i+1})$ is already defined. For this j, we define the following π_1, π_2.

Definition 4. (Definition 8 in [3]). π_1 is the state in $SP_p(i)$ such that $S_p(\pi_1) = j$

Definition 5. (Definition 9 in [3]). π_2 is state j if $S_p(i+1) - 1 = j$. Othewise, π_2 is the state in $SP_p(S_p(i+1) - 1)$ such that $S_p(\pi_2) = j$.

Definition 6. (Definition 10 in [3]).
Let lrs be an array of $m+1$ integers such that for each i, $0 \leq i < m$:
.

$$lrs[i+1] = \begin{cases} 0, & \text{if } S_p(i+1) = 0, & \cdots(1) \\ lrs[\pi_1] + 1, & \text{if } \pi_2 = S_p(\pi_1), \text{ and} & \cdots(2) \\ \min\{lrs[\pi_1], lrs[\pi_2]\} + 1 & \text{otherwise.} & \cdots(3) \end{cases}$$

$lrs[0]$ is set to 0.

The value of $lrs[i]$ is defined as above is not exactly $|repet_p(i)|$ but it is an approximate value of $|repet_p(i)|$. The construction of $lrs[i]$ is linear in space, since each value of this array can be stored in constant space. The two first case (1) and (2) of Definition 6 are computed in constant time. The only problem from third case. In this case, since we have to know π_2 we follow suffix links from $S_p(i+1) - 1$ until π_2 find. For this part we propose a new method to compute π_2. This method compute π_2 from j and p_{i+1} in $O(1)$ instead of following suffix link from $S_p(i+1) - 1$.

Definition 7. For any external transition $\delta(k, \sigma)$ on $Oracle(p)$,
$etbrother(k, \sigma) = l \overset{\text{def}}{\Leftrightarrow} (S_p(l) = k) \wedge (\delta(k, \sigma) = \delta(l, \sigma))$

We will proof that $etbrother(j, p_{i+1}) = \pi_2$(j is the state such that $j = S_p(\pi_1)$).

Lemma 1. For each step $i+1(1\ldots|p|-1)$ of $Oracle(p)$ construction, let j the state such that $j=\pi_1$, we have $etbrother(j,p_{i+1})=\pi_2$.

Proof. Let $etbrother(j,p_{i+1}) = l$. This means $\delta(j,p_{i+1}) = \delta(l,p_{i+1})$ by definition of $etbrother$. Now let $\delta(j,p_{i+1})=q$. Since $j=S_p(\pi_1)$, $\delta(j,p_{i+1}) = S_p(i+1)$. Thus we have $q=S_p(i+1)$. On the other hand $\delta(l,p_{i+1})=q$ since $\delta(j,p_{i+1})=\delta(l,p_{i+1})$. Hence $\delta(l,p_{i+1})$ is constructed at step q of $Oracle(p)$ construction. This means that l is in $SP_p(q-1)$ by construction of factor oracle, that is $l \in SP_p(S_p(i+1)-1)$. Futhermore we have $j=S_p(l)$ by definition of $etbrother$. Since this is exactly the definition of π_2, we have $l=\pi_2$. Thus we have $etbrother(j,p_{i+1})=\pi_2$. $\square$

When we compute $lrs[i+1]$ in the third case, for finding π_2 we only have to search $etbrother(j,p_{i+1})$. The computation of $etbrother$ for each external transition is easy after the external transition is constructed. Figure 2 shows the pseudo-code for the computation of the factor oracle of a given word p together with the table of lrs using the function $etbrother$.

Theorem 1. The complexity of *OracleAndLrs2*$(p=p_1p_2\ldots p_m)$ is $O(m)$ in time and space.

Proof. In [1](Theorem 2) it is proved that the construction of *Oracle*(p) is linear time and space in $|p|$. Clearly a table for lrs needs linear space. Also in the two first case (1) and (2) of Definition 6, each $lrs[i]$ can be computed in constant time. Therefore, we only have to consider the parts of computing $lrs[i]$ for the third case. In this case, the problem is to find π_2. However we can find π_2 from $etbrother(j,p_{i+1})$ by Lemma 1 in constant time. The computation of $etbrother$ for each external transition is also constant time, and the total number of external transitions is at most $m-1$. Hence the complexity of *OracleAndLrs2*$(p=p_1p_2\ldots p_m)$ is $O(m)$ in time and space. $\square$

Figure 3 shows the example of computation of lrs. In this example, $lrs[3]$, $lrs[8]$ and $lrs[12]$ is computed by third case of Definition 6. In the case of $lrs[8]$, π_1 is state 7 and $j=2$ (since j is the state such that $j=S_p(\pi_1)$). Then by Lemma 1 π_2 can be computed as $etbrother(j,p_8)$, which is $etbrother(2,‘c')=3$.

OracleAndLrs2 $(p = p_1 p_2 \cdots p_m)$
  create $Oracle(\epsilon)\{$
    one single state 0
    $S_\epsilon(0) = -1$
  $\}$
  for$(i = 0; i < m; i++)\{$
    $Oracle(p[1 \dots i]) \leftarrow$ NewAddLetter2$(Oracle(p[1 \dots i], p_{i+1}))$
  $\}$
  return $Oracle(p)$ **and** lrs**.**

NewAddLetter2$(Oracle(p[1 \dots i], \sigma))$
  create a new state $i+1$
  create a new internal transition $\delta(j, \sigma) \leftarrow i+1$.
  $j \leftarrow S_p(i)$
  $k \leftarrow i$
  while$(j > -1$ and $\delta(j, \sigma)$ is undefined$)\{$
    create a new external transition $\delta(j, \sigma) \leftarrow i+1$
    $etbrother(j, \sigma) \leftarrow k$ (k is the state such that $S_p(k) = j$)
    $k \leftarrow j$
    $j \leftarrow S_p(j)$
  $\}$
  $\pi_1 \leftarrow k$.
  if $(j = -1)$ $s \leftarrow 0$
  else $s \leftarrow \delta(j, \sigma)$
  $S_p(i+1) \leftarrow s$
  compute $lrs[i+1]$ according to Definition 6.
    (In third case, $\pi_2 \leftarrow etbrother(j, p_{i+1})$.)
  return $Oracle(p[1 \dots i]\sigma)$

Figure. 2: Algorithm: OracleAndLrs2

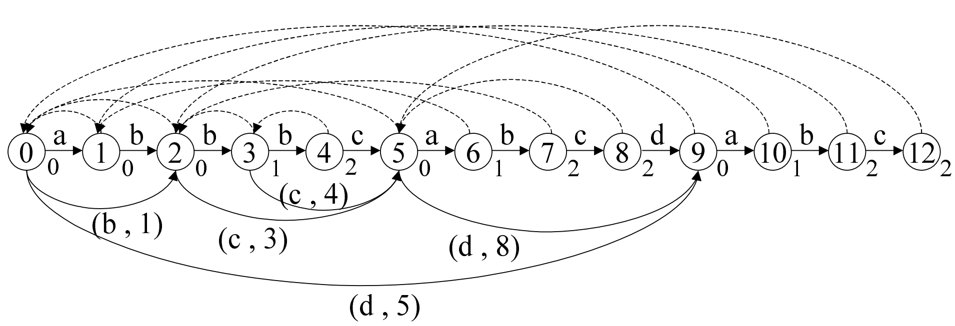

Figure. 3: Example of computation of lrs. The dot arrows represent the suffix link and the plain arrow represent the transitions. The values written on the bottom-right of the states is the lrs values. The pairs written on the external transition represent that the left-value is transition letter and right-value is $etbrother$ values.

References

[1] M. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. *Theory and Practice of Informatics* number 1725, $291 - 306, 1999$.

[2] M. Allauzen, M. Crochemore, and M. Raffinot. Efficient experimental string matching by weak factor recognition. *Lecture Notes in Computer Science, 2089*, $51 - 72, 2001$.

[3] A. Lefebvre, T. Lecroq. Computing repeated factors with a factor oracle, *In Proc. of the* $11th$ *Australasian Workshop on Combinatorial Algorithms* $339 - 348, 2000$.

[4] A. Lefebvre, T. Lecroq. Compror: on-line lossless data compression with a factor oracle, *Information Proceedings Letters volume 83*, $1 - 6, 2001$.

[5] A. Lefebvre, T. Lecroq, H. Dauchel and J. Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes, *bioinformatics volume19 no.3* $319 - 326, 2003$.

[6] Alban Mancheron and Christophe Moan. Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles *In Procedings of the Prague Stringology Conference '04*, $139 - 153, 2004$.

[7] L. Cleophas, G. Zwaan and B. Watson. Constructing Factor Oracles, *In Proceedings of the 3rd Prague Stringology Conference*, 2003.