ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Error analysis of factor oracles

Hisashi Iwasaki

December 2005, C–217

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Error analysis of factor oracles

Hisashi Iwasaki
C/O Prof. Osamu Watanabe,
Dept. of Information Science, School of Science,
Tokyo Institute of Technology
email: watanabe@is.titech.ac.jp

Research Report $C - 217$

abstract

Factor oracles [1] constructed from a given text are deterministic acyclic automata accepting all substrings of the text. Factor oracles are more space economical and easy to implement than similar data structures such as suffix tree[6]. There is, however, some drawback; a factor oracle may accept strings not in the text, which we call a *error acceptance*. In this paper, we charactrize factor orales that accept nonsubstrings erroreously. Using this characterization, we propose an algorithm to decide whether a factor oracle makes an error acceptance during its linear time construction.

keyword: Factor oracles, Error acceptance, Error detection algorithm

1 Introduction

Several data structures, such as suffix tree, suffix automaton, have been developed for finding substrings in a text efficiently. These structures are also used to find all occurrences of a pattern in a text. Factor oracle is one of such data structures proposed by Allauzen et al. [1]. Factor oracle has several advantages. First, factor oracles, which are acyclic automata, are easy to construct; there is a simple algorithm [1] that, for any given text string p, constructs a factor oracle $\mathsf{Oracle}(p)$ for p in *on-line* and within linear time and space in the length of p. Secondary, factor oracles need less space than the other data structures. There is , however, some disadvantage of

using factor oracle; some factor oracles accept strings not in the text, which we call an *error acceptance.*

In this paper we characterize factor oracles that make an error acceptance. More specifically, we give a necessary and sufficient condition such that the on-line factor oracle construction algorithm creates a factor oracle making an error acceptace for the first time. By using this condition, we propose a modification of the algorithm so that it can check whether a constructed factor makes an error acceptance, while increasing computation time only some constant factor.

In the following section, we give necssary definitions. In section 3, we state a characterization for error acceptance and propose an algorithm to check whether a factor oracle makes an error acceptance. Finally, in Section 4, we state conclusion and remarks.

2 Factor oracle

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 $p = p_1 p_2 \ldots p_m$ 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 $p[i \ldots j]$ to denote the substrings of p appearing from the ith character to the jth character. We denote by $\mathit{suf}(i)$ the set of all the suffixes of $p[1 \ldots i]$ for $1 \leq i \leq m$

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.

For example, a factor oracle $\mathit{Oracle}(p)$ for $p = abbbaab$ is given as figure1.

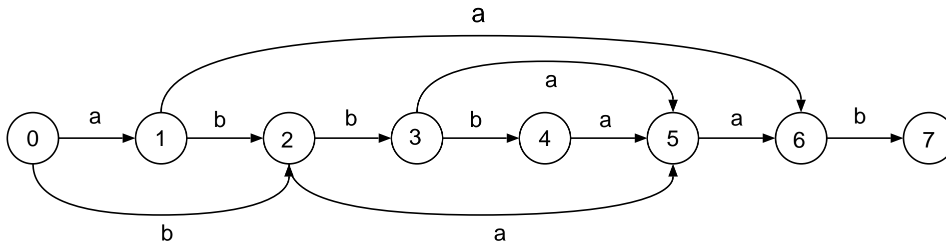

Figure. 1: $Oracle(abbbaab)$

Here state 0 is the initial state. On this figure, the reader can check that

```
Oracle-Sequential (p = p[1...m])
 create Oracle(ε);
     create State 0;
     S_ε(0) ← −1;
 for(i = 1; i ≤ m; i++) {
     Oracle(p[1...i]) ← AddLetter(Oracle(p[1...i − 1], p_i));
 }
```

Figure. 2: Algorithm Oracle-Sequential

```
AddLetter (Oracle(p = p[1...i]), σ)
 create new state i + 1;
 create new transition δ(i, σ) = i + 1;
 j ← S_p(i);
 while(j > −1 and δ(j, σ) is undefined) {
     create new transition δ(j, σ) = i + 1;
     j ← S_p(j);
 }
 if(j = −1)  then s ← 0;
 else s ← δ(j, σ);
 S_pσ(i + 1) ← s;
 return Oracle(p = p[1...i]σ)
```

Figure. 3: Constructing $Oracle(p\sigma)$ from $Oracle(p)$ and σ

it accepts all substrings of p; it is also easy to check that this factor oracle accepts "aba", nonsubstring of p; that is , it makes an error acceptance.

In this paper, we will make use of the notions and properties on factor oracles listed below. The precise definition of facotor oracle is not so important. On the other hand, we define the notion of "factor oracle" by stating an algorithm (Figure 2) for constructing $\mathsf{Oracle}(p)$ from p; this algorithm is one of the two algorithm given in [1], and it constructs an oracle *on-line* reading p from the left to right, and the algorithm runs within linear time and space in $|p|$.

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

For example, in Figure 1, $\text{repet}_p(1) = \epsilon$, $\text{repet}_p(4) = bb$, $\text{repet}_p(7) = ab$.

Definition 2. A function S_p maps each state $i > 0$ of $\mathsf{Oracle}(p)$ 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 $\mathsf{Oracle}(p)$.

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, $SP_p(i) = \{k_0 = i, k_1 = S_p(i), \ldots, k_t = 0\}$.

Proposition 1. (Corollary 4 in [1]) Let $SP_p(i) = \{k_0 = i, k_1 \ldots, k_t = 0\}$ be the suffix path of $p[1 \ldots i]$ in $Oracle(p)$ and let $w_j = \text{repet}_p(k_{j-1})$ for $1 \leq j \leq t$ and $w_0 = p[1 \ldots i]$. Then, for $0 < l < t$, w_l is a suffix of all the w_j, $0 \leq j < l \leq t$.

3 Error acceptance of factor oracles

In this section, for a given string p we consider whether *Oracle*(p) makes an error acceptance.

First, we give a necessary and sufficient condition for checking whether *Oracle*(p) makes an error acceptance in the course of its online construction. Next, we show an algorithm to decide error acceptance of *Oracle*(p) by this condition; the algorithm uses heuristic algorithm to compute $|repet_p(i)|$ given in [3].

3.1 Condition for the first error acceptance of **Oracle**(p)

Before we state a necessary and sufficient condition for an error acceptance, we explain the notion of the first error. For a string $p = p[1 \ldots m]$, we say that $Oracle(p[1 \ldots i-1])$ is *error free (up to state* $i-1$*)* if it accepts only substrings of $p[1 \ldots i-1]$. We say that the *first error acceptance occurs at state* i if $Oracle(p[1 \ldots i-1])$ is error free and $Oracle(p[1 \ldots i])$ *does* accept a string not substring of $p[1 \ldots i]$. This error acceptance is simply called *the first error.* Note that once error acceptance occurs at some state i (i.e. by $Oracle(p[1 \ldots i])$), then all following factor oracles $Oracle(p[1 \ldots i+1]), \cdots, Oracle(p[1 \ldots m])(= Oracle(p))$ keep making some error acceptance. (Let w be the first error accepted by $Oracle(p[1 \ldots i])$. Then, for every $j > i$, a string $wp_i \ldots p_j$ is accepted erroreously by $Oracle(p[1 \ldots j])$.) Thus, for checking whether *Oracle*(p) makes an error acceptance, it suffices to find the first error. The following theorem states a neccessary and sufficient condition that the first error occurs in *Oracle*(p).

Theorem 1. Assume that $Oracle(p[1 \ldots i-1])$ is error free. Let $u = \text{repet}_p(i-1)$, and let $L(i)$ be a set of strings accepted at state i. Let $\sigma = p_i$. Then, for any $v \in L(S_p(i-1))$, we have

$v\sigma$ is the first error acceptance
$\Leftrightarrow \exists\ v \in L(S_p(i-1))$ such that
(1) $|v| > |u|$, and
(2) $\delta(S_p(i-1), \sigma) = i$.

Proof. ($\Leftarrow$) Let v be a string satisfying the condition (1) of the theorem. We should notice that strings accepted correctly at state i are only suffixes of $p[1 \ldots i]$. So if v is not in *suf*$(i-1)$, $v\sigma(= vp_i)$ is not in *suf*(i). Therefore, $v\sigma(= vp_i)$ is accepted erroreously at state i. Hence our goal is to show

$v \notin \mathit{suf}(i-1)$. But we can easily show that $v \notin \mathit{suf}(i-1)$, if v were in $\mathit{suf}(i-1)$, then $\mathrm{repet}_p(i-1)$ shuould not be u but v since $|v| > |u|$, which contradicts our definition of u. Thus, we have $v \notin \mathit{suf}(i-1)$.

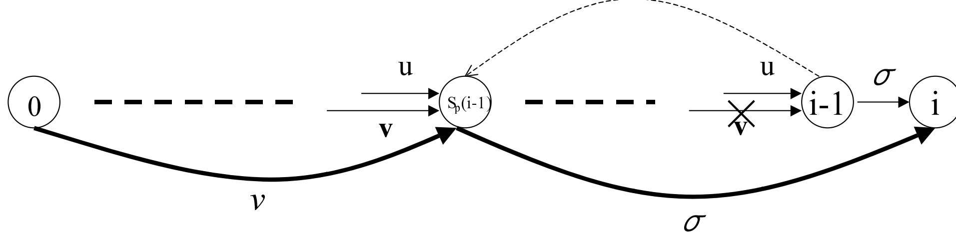

Figure. 4: factor oracle that $v\sigma$ is error-accepted

($\Rightarrow$) Suppose that $v\sigma$ is accepted erroreously at state i. This means that $v\sigma \notin \mathit{suf}(i)$ and $\delta(i-1, \sigma) = i$. We only have to show that $|v| > |u|$. Assume to the contrary that $|v| \leq |u|$. Then $v \in \mathit{suf}(i-1)$ since the string v is suffix of u. On the other hand, we know $v \notin \mathit{suf}(i-1)$ because $v\sigma \notin \mathit{suf}(i)$ and $\delta(i-1, \sigma) = i$. We reach a contradiction, thus, $|v| > |u|$.

□

This theorem gives us a way to check the first error acceptance occurs at $Oracle(p[1 \ldots i])$ when ith symbol p_i is added to $Oracle(p[1 \ldots i-1])$. That is, first search for a string v such that $|v| > |repet_p(i-1)|$ which is one of the condition of Theorem 1; then check whether an external transition is defined to the state i. In the following sections , we discuss the way to achieve this test efficiently during the on-line construction of $\mathit{Oracle}(p)$.

3.2 Computing repeated suffix for each prefix

In [3] a heuristic algorithm to find longest repeats using factor oracles was proposed. The length of a repeated suffix of $p[1 \ldots i]$, denoted by $lrs[i]$, computed in this algorithm is defined reccursively as below (Definition 6). This algorithm also builds $\mathit{Oracle}(p)$ and compute $lrs[i]$ for every i, $0 < i \leq |p|$, the complexity is $O(|p|)$ in time and space.

During the construction of $Oracle(p[1 \ldots i+1])$ from $Oracle(p[1 \ldots i])$ 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, $0 \leq i < m$:
.

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

$lrs[0]$ 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 $|repet_p(i)|$. Thus, we can't use $lrs[i]$ instead of $|\mathrm{repet}_p(i)|$ in Theorem 1. However, in the following lemma we will prove that $lrs[i] = |repet_p(i)|$ for any i provided that $Oracle(p[1\dots i])$ doesn't make an error acceptance. This enable us to use $lrs[i]$ instead of $|repet_p(i)|$ in Theorem 1.

Lemma 1. Provided that $Oracle(p[1\dots l])$ is error free, for any i, $0 \leq i < l$, we have
$(*)$ $\quad lrs[i] = |repet_p(i)|$.

Proof. We prove $(*)$ by induction on the number i of states constructed by the on-line construction algorithm. At the initial step, where $i = 0$, we have by definition that $lrs[0] = 0$ and that $|repet_p(0)| = |\epsilon| = 0$; hence, $(*)$ holds. For the induction step, we assume that $lrs[k] = |repet_p(k)|$ for all k, $0 \leq k \leq i$, and consider the step where the state $i+1$ is constructed. The case $S_p(i+1) = 0$ is easy because in this case where $|repet_p(i+1)| = |\epsilon| = 0$ and $lrs[i+1] = 0$ by definition. Thus we consider the case that $S_p(i+1) = q$ for some $q \neq 0$. There are two cases depending on whether $q - 1 = S_p(\pi_1)$ or not.

First consider the case that $q-1 = S_p(\pi_1)$. In this case since $\pi_2 = S_p(\pi_1)$ (by definition of π_2), we have $lrs[i+1] = lrs[\pi_1] + 1$. This means that an internal transition $\delta(q-1, p_{i+1}) = q$ is constructed when the state $i+1$ is constructed. Hence the length of the longest repeated suffix $|repet_p(i+1)|$ have only to simply add 1 to $|repet_p(\pi_1)|$, that is, $|repet_p(i+1)| = |repet_p(\pi_1)| + 1$. On the other hand, by induction hypothesis, we have $|repet_p(i+1)| = lrs[\pi_1] + 1 = lrs[i+1]$. From these claims $(*)$ follows.

Now consider the case that $q - 1 \neq S_p(\pi_1)$. In this case $lrs[i+1] = \min\{lrs[\pi_1], lrs[\pi_2]\} + 1$ by definition. Let j be the state such that $S_p(\pi_1) = j$. Then the transition by p_{i+1} from state j to q is an external transition. Since we assume that $Oracle(p[1\dots q])$ is error free, both strings $p[1\dots j]p_{i+1}$ and $p[1\dots \pi_2]p_{i+1}$ is suffix of $p[1\dots q]$. This implies that $p[1\dots j]$ is suffix of $p[1\dots \pi_2]$. Note here that the longest repeated suffix $repet_p(\pi_2)$ is accepted at state j and $p[1\dots j]$ is indeed the longest string accepted at state j. Hence

we have $repet_p(\pi_2) = p[1 \ldots j]$; furthermore the relation $lrs[\pi_1] \leq lrs[\pi_2]$ holds since $lrs[\pi_1]$ is also at most j. Then by definition of lrs, we have $lrs[i+1] = lrs[\pi_1] + 1$. Also $lrs[i+1] = |repet_p(\pi_1)| + 1$ by induction hypothesis. Thus, for $(*)$ it suffices to prove $repet_p(i+1) = repet_p(\pi_1)p_{i+1}$ which is our goal below.

Let u, v be a string such that $repet_p(i+1) = up_{i+1}(u \in \Sigma^*)$, $v = repet_p(\pi_1)$, and our goal is to show $u = v$. The string u appears at state j from $S_p(i+1) = q$ and $\delta(j, p_{i+1}) = q$. The string v also appears at state j. Any strings accepted at state j is suffix of $p[1 \ldots j]$ since $Oracle[1 \ldots j]$ is error free by assumption. Thus we have $u, v \in \mathit{suf}(j)$. Moreover u is suffix of $p[1 \ldots i]$ by definition, v is also suffix of $p[1 \ldots i]$ from Proposition 1; that is, $u, v \in \mathit{suf}(i)$. We want to verify $u = v$. Consider now the following two cases; $|u| < |v|$ and $|u| > |v|$, and lead to a contradiction. In the first case, $|u| < |v|$, $repet_p(i+1)$ must be vp_{i+1} since vp_{i+1} appears at least twice at state q and $i+1$, $|vp_{i+1}| > |up_{i+1}|$. This contradicts $repet_p(i+1) = up_{i+1}$. In the second case, $|u| > |v|$, let $SP_p(i) = \{j_0 = i, j_1, \ldots, j_t = j, \ldots, j_s = 0\}$ be the suffix path of $p[1 \ldots i]$. Then there exists an integer $l(1 \leq l \leq t)$ such that $|repet_p(j_l)| \leq |u| < |repet_p(j_{l-1})|$. This means $S_p(j_l) = j$ and $repet_p(j_l) = u$. That is, $j_{l+1} = j$ and $j_l = \pi_1$; $repet_p(j_l) = repet_p(\pi_1) = v = u$. This contradicts $|u| > |v|$. Since we lead a contradiction from both cases, we have $|u| = |v|$ which implies $u = v$. Thus $repet_p(i+1) = up_{i+1} = vp_{i+1} = repet_p(\pi_1)p_{i+1}$ is verified. □

3.3 Error acceptance detection algorithm

We build an algorithm to decide whether *Oracle*(p) makes an error acceptance using Theorem 1 and Lemma 1. The algorithm is **FOError** (Figure. 5), we explain this algorithm along this figure. As an example, we state the execution of the algorithm for construction of $Oracle(abbbc)$ in Figure. 7.

The variable flag may take one of values in the set $\{0, 1, 2\}$. At first the variable flag is initialized 0. This algorithm constructs an oracle *on-line* reading p from the left to right. The function **NewAddLetter** in for-statement computes $Oracle(p[1 \ldots i])$ and $lrs[i]$; this function is exactly the same function given in [3]. After these computation, we check the conditions of Theorem 1. First, we check the condition (1) of Theorem 1; for a string v is accepted at state $S_p(i-1)$ and $u = \mathrm{repet}_p(i-1)$, whether $|v| > |u|$ or not. Since the longest string is accepted at state $S_p(i-1)$ is $p[1 \ldots S_p(i-1)]$, we can use $S_p(i)$ as $|v|$. Moreover we can replace $|u|$ by $lrs[i-1]$ using Lemma 1. Hence we check whether $S_p(i-1) > lrs[i-1]$ or not. If $S_p(i-1) > lrs[i-1]$, we change the variable flag from 0 into 1. Next, we check the condition (2) of Theorem 1, that is whether external transision is defined from $S_p(i-1)$ to i or not. Notice that once the variable flag becomes 1, the condition (1) keeps satisfied until an error acceptance occurs. Thus once the value of flag

```
FOError (p = p_1 p_2 ··· p_m)
create Oracle(ε){
  one single state 0
  S_ε(0) = −1, flag ← 0
}
for(i = 1; i ≤ m; i++){
   Oracle(p[1...i]) ← NewAddLetter(Oracle(p[1...i−1], p_i))
   if(flag == 0){
     if(S_p(i−1) > lrs[i−1])
       flag ← 1 .
   }
   else if(flag == 1){
     if(an external transition is defined to state i)
       flag ← 2 ··· (error-acceptance).
   }
}
return Oracle(p) and flag.
```

Figure. 5: Algorithm: FOError

is 1, we check only the condition (2) whenever the factor oracle is updated by adding a letter. If the condition (2) is satisfied with flag= 1, the value of flag becomes 2. Then two conditions of Theorem 1 is satisfied and the factor oracle makes an error acceptance. So if the value of flag isn't 2 when the construction of *Oracle*(p) is finished, *Oracle*(p) doesn't make an error acceptance, that is error free. If we halt the algorithm at the stage when the value of flag is equal to 2, we can decide at the stage whether *Oracle*(p) makes an error acceptance.

Theorem 2. Algorithm **FOError**$(p = p_1 p_2 \cdots p_m)$ computes *Oracle*(p) and $lrs[i] (\forall i, 0 \leq i \leq m)$. In addition, it decides whether *Oracle*(p) makes an error acceptance.

Proof. The correctness of *Oracle*(p) and $lrs[i]$ is proved in [3]. Using Lemma 1 and Theorem 1, we can also prove about judgement of error acceptance. □

Theorem 3. The complexity of **FOError**$(p = p_1 p_2 \ldots p_m)$ 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 $|p|$. [3](Theorem 2) proves that the construction of $lrs[1 \ldots m]$ is linear time and space in $|p|$ [1]. Thus we only have to prove the parts for error acceptance test (line 8-15, Figure 5). For each i, the number

[1]Detail proof and implementation are in preparation [13].

```
NewAddLetter(Oracle(p[1...i], σ)){
  create a new state i + 1
  δ(i, σ) ← i + 1
  j ← S_p(i)
  π_1 ← i
  while(j > −1 and δ(j, σ) is undefined){
     δ(j, σ) ← i + 1
     π_1 ← j
     j ← S_p(j)
  }
  if (j = −1) s ← 0
  else s ← δ(j, σ)
  S_p(i + 1) ← s
  compute lrs[i + 1] according to Definition 6.
  return Oracle(p[1...i]σ)
}
```

Figure. 6: Function NewAddLetter

of if-statements is at most four times. Therefore, during the construction of *Oracle*(p), the number of executed comparisons is at most $4m$. Hence the complexity of **FOError**$(p = p_1p_2 \dots p_m)$ is $O(m)$ in time and space.

□

4 Conclusions

We analyze the situation that factor oracles accept a string which is not substrings of a given text p. We obtained a necessary and sufficient condition of the first error in the process of constructing *Oracle*(p). Moreover using the condition, we provided a method to decide whether *Oracle*(p) makes an error acceptance in time and space $O(|p|)$.

Our motivation of this study is that factor oracles accept a string not in a given text and what this strings is like. We made the first error clear, but we don't know the language is accepted erroreously by *Oracle*(p). In [8], a characterization of the language recognized by factor oracles is described. Also, other questions stay open about factor oracle. For example, the factor oracle is not minimal considering the number of transitions among the automaton of $m + 1$ states which recognize at least the factors. Does there exist an algorithm to build this reduced automaton? This remains an open problem.

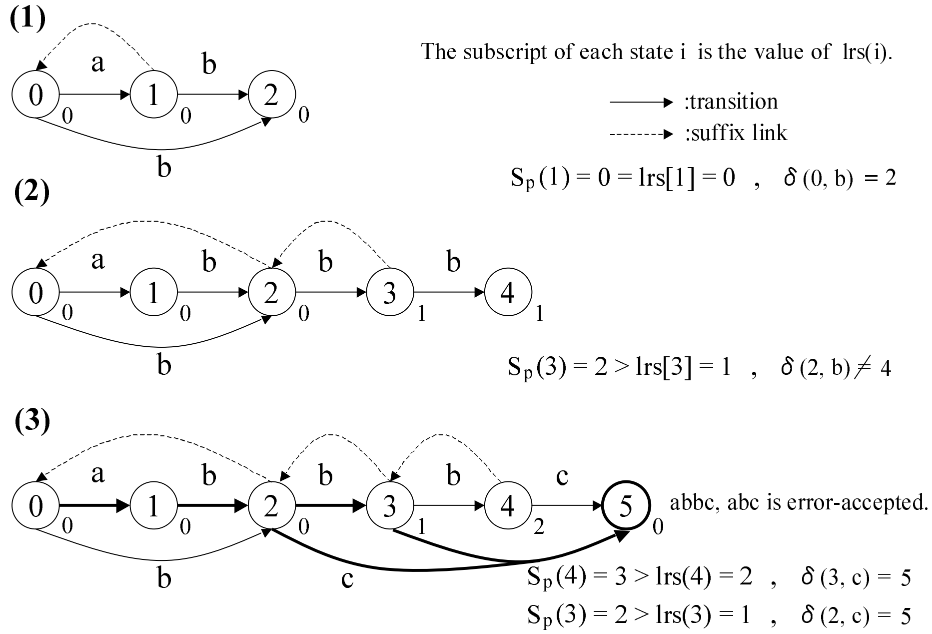

Figure. 7: The behavior of our algorithm for constructing $Oracle(abbbc)$

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] Edward M. McCreight. A space-economical suffix tree constrction algorithm, *Journal of the ACM* $23 : 262 - 272, 1976$.

[7] U. Manber, G. Myers. A new method for on-line string searches, *1st Annual ACM-SIAM Symposium on Discrete Algorithms* 319−327, 1990.

[8] 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.

[9] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen, and J. Selferas. The smallest automaton recognizing the subwords of a text. *Theoretial Computer Science, volume 40*, 31 − 55, 1985.

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

[11] U. Manber, G. Myers. A new method for on-line string searches, *1st Annual ACM-SIAM Symposium on Discrete Algorithms* 319−327, 1990.

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

[13] H. Iwasaki and O. Watanabe. Detail construction of computing a repeated factor. to appear.