THE STRUCTURE OF FACTOR ORACLES[*][†]

MAXIME CROCHEMORE[‡]

Institut Gaspard-Monge, Université de Marne-la-Vallée
F-77454 Marne-la-Vallée, Cedex 2, FRANCE
and
Department of Computer Science, King's College London, London WC2R 2LS, UK
mac@univ-mlv.fr

LUCIAN ILIE[§][¶] EMINE SEID-HILMI

Department of Computer Science, University of Western Ontario
N6A 5B7, London, Ontario, CANADA
{*ilie,eseidhil*}*@csd.uwo.ca*

The factor oracle is a relatively new data structure for the set of factors of a string. It has been introduced by Allauzen, Crochemore, and Raffinot in 1999. It may recognize nonfactors (hence the name "oracle") but its implementational simplicity and experimental behaviour are stunning; factor oracle based string matching has been conjectured optimal on average. However, its structure is not well understood. We take important steps in clarifying its structure by explaining how it can be obtained as a quotient of the trie of the set of factors. When seen this way, all known properties of the factor oracle become simple observations. Also, we introduce a framework where various oracles can be compared. The factor oracle is better than several natural ones obtained from the trie of the set of factors, the suffix and the factor automata, respectively.

Keywords: factor oracle; string matching; suffix trie; suffix tree; graph quotient

MSC: 68P05, 68P10, 68P20, 68Q45

1. Introduction

The factor oracle is a relatively new data structure for the set of factors of a string which has been introduced by Allauzen, Crochemore, and Raffinot in [1, 2]. The starting point was the notion of *weak factor recognition* which means constructing a NO-biased algorithm for detecting factors of a string. (That is, when testing whether a string is a factor, a NO answer is always correct.) In the string matching algorithms based on reversed factors, identifying correctly nonfactors is enough. Therefore, the factor oracle recognizes all factors of a string but may recognize some nonfactors as

[*]An abstract of this work has been presented at the 11th CIAA, Taipei, 2006; see [8].
[†]This work has been completed during the second author's stay at Institut Gaspard-Monge.
[‡]Research supported in part by CNRS.
[§]Corresponding author
[¶]Research supported in part by NSERC and CNRS.

well (hence the name "oracle"). On the other hand, the string-matching algorithms based on it are as efficient as the best existing ones but far simpler to implement; they also require less memory. According to the experimental results, it has been conjectured in [1, 2] that these algorithms are optimal on average. A number of other applications of the factor oracle to data compression, repetitions searching, and learning have been investigated in [3, 9, 10, 11, 12].

The structure of the factor oracle is however not well understood. Proving various properties of it was, so far, rather difficult and therefore solving the open problems concerning it difficult to attempt.

We present here a different way of looking at the factor oracle, namely as a quotient of the trie for the set of factors. Using our construction, all known properties of the factor oracle become simple observations. Moreover, we introduce the general notion of an oracle for the set of factors of a string – the factor oracle is a particular case here – and build a framework for comparing such oracles since, arguably, all of them have to include a quotient of the trie.

Several other natural oracles can be obtained in this way and we prove that the factor oracle is the best among those. Particular examples exist when the factor oracle can be improved but whether there exists a general strategy for building better oracles remains open.

We hope that the new approach will be of help in solving various open problems concerning the factor oracle; see Section 9 for details.

The paper is structured as follows. We recall in the next section all basic concepts needed and then present in Section 3 a variant of Ukkonen's algorithm for building tries in which some additional information is computed; this information helps us later in constructing quotients of the trie. The very simple algorithm of [1, 2] for constructing the factor oracles is described briefly in Section 4. Section 5 describes an oracle naturally obtained from the trie, called trie oracle. In Section 6 we show how the factor oracle can be obtained from the trie oracle and why it is better. Two other natural oracles can be obtained from the suffix and the factor automata, respectively. We show in Section 7 that they are the same as the trie oracle and therefore never better than the factor oracle. Section 8 contains a direct construction of the factor oracle as a quotient of the trie, which makes it very simple to prove things about the factor oracle as done in Section 9. We conclude with a brief discussion in Section 10.

2. Basic definitions

Let A be an alphabet; A^* is the free monoid generated by A, that is, the set of all finite strings over A. The empty string is ε. For a string $w \in A^*$, we denote by $|w|$ the length of w. If $w = xyz$, for $w, x, y, z \in A^*$, then x, y, z are a *prefix*, *factor*, and *suffix* of w, resp. When different from w they are called *proper*. The set of all factors (suffixes) of w is denoted $\mathsf{Fact}(w)$ ($\mathsf{Suff}(w)$, respectively).

For a string w, we shall denote by $\mathsf{suf}(w)$ the longest proper suffix of w, that is the string obtained from w by removing its first letter; for the empty string we have $\mathsf{suf}(\varepsilon) = \mathsf{nil}$. The ith letter of w is $w[i]$ and, for $1 \leq i \leq j \leq |w|$, we denote $w[i..j] = w[i]w[i+1]\cdots w[j]$.

A *finite automaton* is a directed graph. In an automaton the nodes are usually called *states* and the labelled edges are called *transitions.* where the edges are labelled by letters from A; if we have an edge $i \xrightarrow{a} j$, then j is an *a-son* of i.

The automaton is *deterministic* if any node has at most one a-son, for any letter a and *nondeterministic* otherwise. To define the language recognized by an automaton, we need to identify an initial node and some final nodes. Then, the strings recognized are precisely those labelling paths from the initial node to a final node. The set of the strings recognized by an automaton M is denoted $L(M)$. In general, for a node i, the language $L(i)$ is the set of all labels of paths starting from i and ending in a final node. Unless otherwise specified, all our graphs, when seen as automata, will have 0 or ε as initial node and all nodes are final.

The *quotient* of a graph G is any graph obtained from G by merging together the nodes according to a given equivalence relation $\equiv$. The edges are modified accordingly, so that multiple edges labelled the same between two nodes are eliminated. The quotient is denoted $G|_{\equiv}$.

Inspired by the discussion of [1, 2] on the properties the factor oracle should have, we introduce the notion of *oracle for the set of factors of a string w*; it is a deterministic automaton which:

(o_1) recognizes at least all factors of w;
(o_2) is acyclic (it recognizes a finite set of strings);
(o_3) has $|w| + 1$ states (lowest possible);
(o_4) has linearly many edges (independent of alphabet size);
(o_5) for each node, all incoming edges have the same label (for efficient implementation).

The criteria (o_1)-(o_4) appear in [1, 2]; (o_5) is new but nevertheless satisfied by the factor oracle; it is very important for implementation because it makes the memorization of the edge labels unnecessary. Notice the difference between an oracle for the set of factor of a string and *the* factor oracle of [1, 2]. As we shall work with both automata and trees, we shall simply call them all graphs.

3. Ukkonen's algorithm for tries

The *trie* of a string $w \in A^*$, denoted $\mathsf{Trie}(w)$, is the tree containing all factors of w. Formally, it is a directed graph having as nodes the factors of w and (labelled) edges $u \xrightarrow{a} ua$, where $u, ua \in \mathsf{Fact}(w)$, $a \in A$. Each factor is the label of a path starting from the root. See Fig. 1 for an example. Whenever we discuss tries, we shall identify each node with the corresponding path from the root.

The trie is, in some sense, the most basic data structure for strings as most of the other ones – suffix trees, DAWGs, suffix automata – can be obtained from it.

As we show below, the factor oracle can also be obtained from it.

Ukkonen [14] gave a linear-time on-line algorithm for constructing suffix trees which are tries with all chains (paths of nodes of outdegree 1) compacted. However, his construction works also for the simpler case of tries. We describe it below as it will be useful in constructing a number of oracles from the obtained trie.

We shall need *suffix links*, which are links from a node u to $\mathsf{suf}(u)$; we shall represent them as dotted arrows; the regular edges are represented as solid arrows. The *suffix path* from a node u is:

$$u, \mathsf{suf}(u), \mathsf{suf}^2(u), \dots ,$$

continuing as long as the suffix links are defined; we shall denote it by $\mathsf{suf}^*(u)$.

The algorithm works sequentially, considering the letters of w one at a time. To add one letter a, we start from the deepest node in the current trie and follow the suffix links adding new a-sons with their suffix links; this is done until one node having an a-son is found or the value of the suffix link becomes nil; see [7] for details.

Important for us later will be the time each node has been created, that is, the index of the letter in the string which caused the addition of that node. This will be denoted, for a node u, by $\mathsf{time}(u)$; we shall sometimes write the time as a subscript to the label of the node: $u_{\mathsf{time}(u)}$.

Here is the pseudocode for Ukkonen's algorithm. We also compute the time values and some S'-links which will be discussed later.

```
UKKONEN_TRIE($w$)
- given a string $w = w[1]w[2]\cdots w[n]$, $w[i] \in A$, $1 \le i \le n$;
- return Trie($w$);
 1.  construct the two-node Trie($w[1]$) with the suffix links
 2.  for $i$ from 2 to $n$ do
 3.      $v \leftarrow$ deepest leaf of Trie($w[1..i-1]$)
 4.      $k \leftarrow \min\{i \mid \mathsf{suf}^i(v)$ has a $w[i]$-son or it is nil$\}$
 5.      for $\ell$ from 0 to $k-1$ do
 6.          create $\mathsf{suf}^\ell(v) \xrightarrow{w[i]} x$
 7.          create a suffix link for $x$    [ to $w[i]$-son of $\mathsf{suf}^{\ell+1}(v)$ (or $\varepsilon$ if nil) ]
 8.          $\mathsf{time}(x) \leftarrow i$
 9.      if $\mathsf{suf}^k(v) =$ nil then $S'(i) \leftarrow 0$
10.      else $u \leftarrow w[i]$-son of $\mathsf{suf}^{k-1}(v)$
11.          $S'(i) \leftarrow \mathsf{time}(u)$
```

Notice that, at step 7, the $w[i]$-son of $\mathsf{suf}^{\ell+1}(v)$ is created if it does not exist.

The trie obtained for the string baababbabc is shown in Fig. 1. The string has been chosen to show the most important aspects of our constructions. Setting, by convention, $S'(0) = -1$, the values of S' for the example in Fig. 1 are:

i	0	1	2	3	4	5	6	7	8	9	10
$S'(i)$	−1	0	0	2	1	2	4	1	2	6	0

The following remarks about Ukkonen's trie construction algorithm are useful.

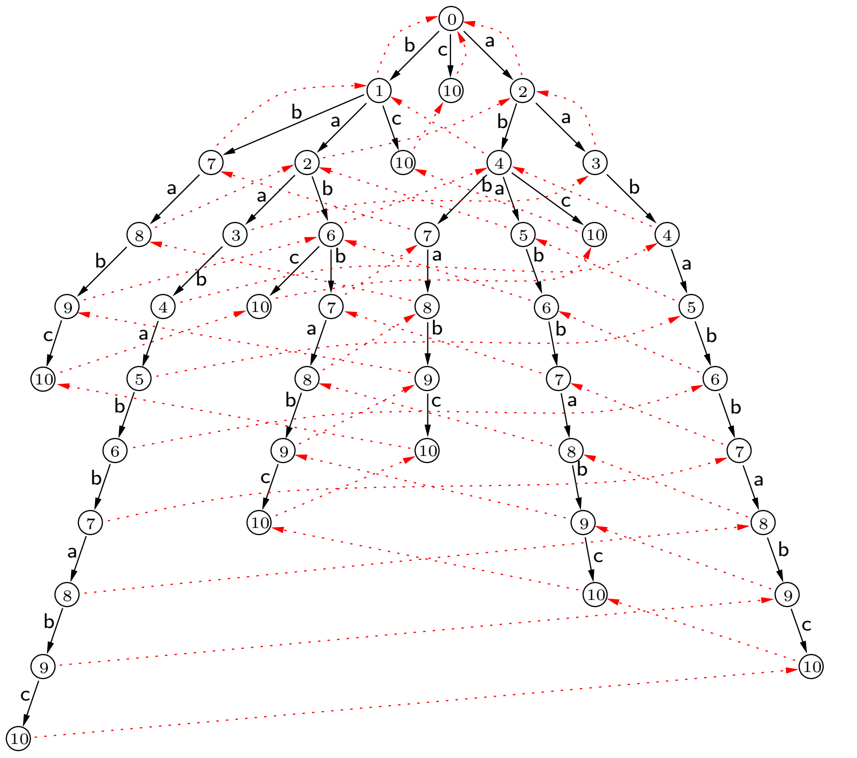

Fig. 1. The trie built by the algorithm UKKONEN_TRIE for the string baababbabc.

Remark 1. If v is the deepest node in the trie with $\mathsf{time}(v) = i$, then all nodes in the trie with time equal to i are found on the suffix path $\mathsf{suf}^*(v)$. The S'-link $S'(i)$, for $i > 0$, is the first node on the suffix path which has a time different from i (or 0 if such a node does not exist).

Remark 2. Notice that, if u with $\mathsf{time}(u) = j$ is a node on the suffix path $\mathsf{suf}^*(v)$, then not all nodes with time equal to j need to be on the suffix path $\mathsf{suf}^*(v)$. In our example we have the suffix path (the subscripts show the time values): $\mathsf{baa_3}, \mathsf{aa_3}, \mathsf{a_2}$, but the node $\mathsf{ba_2}$ is not on the suffix path of baa. The shallowest (closest to root) node with time value j must be on $\mathsf{suf}^*(v)$. This gives also that $u \in \mathsf{suf}^*(v)$ implies $\mathsf{time}(u) \in S'^*(\mathsf{time}(v))$, but the converse need not be true. (Here $S'^*(i)$ is the S'-path of i, that is, $i, S'(i), S'^2(i), \ldots$.)

4. A simple algorithm for the factor oracle

We recall in this section the sequential algorithm of Allauzen, Crochemore, and Raffinot (ACR, for short) for constructing factor oracles. However, we shall not assume we know that the obtained graph is the factor oracle. We shall show later that the same object can be obtained from the trie we described before and the most important properties we need about the factor oracle will follow from there.

```
ACR_FACTOR_ORACLE($w$)
- given a string $w = w[1]w[2]\cdots w[n]$, $w[i] \in A$, $1 \le i \le n$;
- return Factor_Oracle($w$);
1.  $S(0) \leftarrow -1$
2.  for $i$ from 1 to $n$ do
3.      create $i-1 \xrightarrow{w[i]} i$
4.      $\ell \leftarrow S(i-1)$
5.      while ($\ell \neq -1$) and ($\ell$ has no $w[i]$-son) do
6.          create $\ell \xrightarrow{w[i]} i$
7.          $\ell \leftarrow S(\ell)$
8.      if $\ell = -1$ then $S(i) \leftarrow 0$
9.      else $S(i) \leftarrow$ the $w[i]$-son of $\ell$
```

The factor oracle for the string baababbabc is shown in Fig. 2. Again, regular edges are solid arrows whereas the S-links are dotted. Notice the string baabc which is recognized but is not a factor. The S-links for the example are:

i	0	1	2	3	4	5	6	7	8	9	10
$S(i)$	-1	0	0	2	1	2	4	1	2	4	0

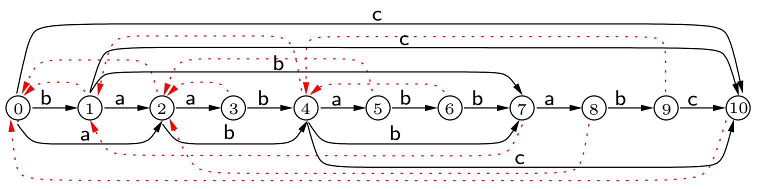

Fig. 2. Factor_Oracle(baababbabc)

5. The trie oracle

We can obtain another oracle for Fact(w) from the Trie(w) in a natural way; we simply merge all nodes with the same time value to obtain Trie(w)$|_{\mathsf{time}}$. Obviously, time gives an equivalence relation on set of nodes. The nodes of Trie(w)$|_{\mathsf{time}}$ are the corresponding time values. The one for our string baababbabc is shown in Fig. 3. The edges are shown as continuous arrows and the S'-links are dotted. There are three differences with respect to the factor oracle – two edges and one S'-link; they are shown in bold.

We notice first that it is nondeterministic; the node 2 has two b-sons. We make it deterministic in the following way: eliminate any edge $i \xrightarrow{a} j$ whenever we can

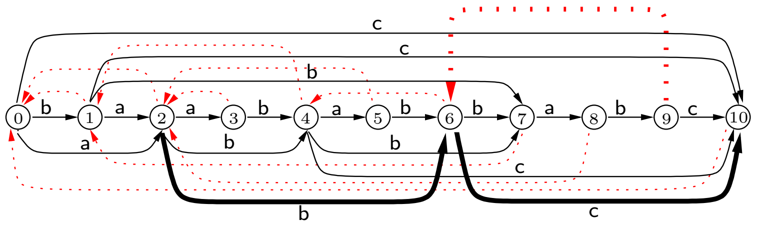

Fig. 3. $\mathsf{Trie}(\mathsf{baababbabc})|_{\mathsf{time}}$

find $i \xrightarrow{a} k$ with $k < j$. In our example the edge $2 \xrightarrow{\mathsf{b}} 6$ is removed. Denote the obtained graph $\mathsf{Trie\_Oracle}(w)$. The one for our example is shown in Fig. 4.

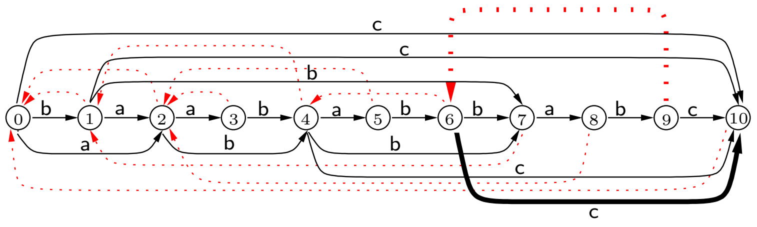

Fig. 4. $\mathsf{Trie\_Oracle}(\mathsf{baababbabc})$

We prove below that the trie oracle is an oracle for $\mathsf{Fact}(w)$. The trie oracle is deterministic, acyclic, has $|w| + 1$ states, and for each node all incoming edges are labelled the same. We prove next that it recognizes at least all factors of w. The next lemma concerns $\mathsf{Trie}\,|_{\mathsf{time}}$ and is useful for our purpose.

Lemma 3. *In* $\mathsf{Trie}(w)|_{\mathsf{time}}$*, if* $j \in S'^*(i)$*, then* $L(i) \subseteq L(j)$*.*

Proof. By induction on i from $|w|$ to 1. For $i = |w|$, we have $L(i) = \{\varepsilon\}$ and the property holds. Assume it true for $i+1, i+2, \ldots, |w|$ and prove it for i. For any edge $i \xrightarrow{a} i'$, there exists $v \xrightarrow{a} va$ in $\mathsf{Trie}(w)$ with $\mathsf{time}(v) = i$ and $\mathsf{time}(va) = i'$. We can also find a node $u \in \mathsf{suf}^*(v)$ such that $\mathsf{time}(u) = j$ and there exists an edge $u \xrightarrow{a} ua$; we can take for u the shallowest node with time value j. Now $ua \in \mathsf{suf}^*(va)$ and if we put $j' = \mathsf{time}(ua)$, then $j' \in S'^*(i')$. By the inductive hypothesis, $L(i') \subseteq L(j')$. As this holds for every a-son of i, the claim follows. $\square$

Corollary 4. *For any* w*,* $\mathsf{Trie\_Oracle}(w)$ *recognizes at least all factors of* w*.*

Proof. It is clear that $\mathsf{Trie}(w)|_{\mathsf{time}}$ recognizes at least all factors of w. But whenever an edge $j \xrightarrow{a} i$ is removed to create $\mathsf{Trie\_Oracle}(w)$, there is another edge $j \xrightarrow{a} \ell$, with $\ell < i$. In such a case we have $\ell \in S'^*(i)$ and Lemma 3 says that eliminating the former edge does not affect the set of recognized strings. □

What is left to show is property (o_4) in the definition of an oracle.

Lemma 5. *For any w, $\mathsf{Trie}(w)|_{\mathsf{time}}$ has at most $2|w| - 1$ edges.*

Proof. There are two types of edges in $\mathsf{Trie}(w)|_{\mathsf{time}}$ and $\mathsf{Trie}(w)$: (type 1) $i \xrightarrow{w[i]} i+1$ and (type 2) $i \longrightarrow j$, with $j \neq i + 1$; for $\mathsf{Trie}(w)$ we consider the time values of the ends of an edge. There are $|w|$ edges of type 1 in $\mathsf{Trie}(w)|_{\mathsf{time}}$ so we need to count the other ones.

In $\mathsf{Trie}(w)$, each internal node has exactly one outgoing edge of type 1. Therefore, the number of edges of type 1 is the same as the number of internal nodes. The total number of edges equals the number of nodes minus one (it is a tree). Therefore, the number of edges of type 2 is the number of leaves minus one. But the number of leaves is at most the number of nonempty suffixes of w, that is, $|w|$. So, there are at most $|w| - 1$ edges of type 2. The claim follows. □

Note that this lemma also applies to $\mathsf{Trie\_Oracle}(w)$.

Corollary 6. *For any w, the set of strings recognized by $\mathsf{Trie\_Oracle}(w)$ is closed under taking factors.*

Proof. The sequence of S'-links starting from any node of $\mathsf{Trie\_Oracle}(w)$ ends in the initial node 0. Therefore, Lemma 3 implies that any string recognized starting from some i is also recognized starting from 0. □

The following proposition follows from the above results.

Proposition 7. *For any w, $\mathsf{Trie\_Oracle}(w)$ is an oracle for $\mathsf{Fact}(w)$.*

6. The factor oracle from the trie oracle

We show in this section how to obtain the factor oracle from the trie oracle by removing certain edges. The differences between the two graphs in Figs. 2 and 4 are shown in bold in the latter. The factor oracle recognizes strictly less strings than the trie oracle in this case; for instance, baababc is recognized by the latter but not by the former. Notice that we do not attempt to find a better algorithm for computing the factor oracle (the simplicity of the one in [1, 2] we presented above seems almost impossible to beat) but to understand its tricky structure and properties. Here is the result.

Theorem 8. *For any string w, we have*

(i) for any i there is $e_i \geq 1$ such that $S'^{e_i}(i) = S(i)$;

(ii) $\mathsf{Factor\_Oracle}(w)$ *is obtained from* $\mathsf{Trie\_Oracle}(w)$ *by removing, for all* i *with* $e_i \geq 2$*, all edges* $S'^k(i) \xrightarrow{w[i+1]} i+1$, $1 \leq k \leq e_i - 1$.

Proof. We prove that the two assertions hold for any prefix of length n of w by induction on n. The assertions are clear for $n = 1$. We assume they hold for n and prove them for $n+1$. Denote $w[n+1] = a$ and consider the path of S-links in $\mathsf{Factor\_Oracle}(w[1..n])$ starting from n and ending in the first node which has an a-son:

$$n_0 = n_{i_0} = n, n_{i_1} = S(n), n_{i_2} = S^2(n), \ldots, n_{i_k} = S^k(n) \ , \tag{4}$$

for some $0 = i_0 < i_1 < i_2 < \cdots < i_k$; that is, all n_{i_ℓ}, $0 \leq \ell \leq k-1$ do not have an a-son but n_{i_k} has one, say m. If there is no node with an a-son on this path, then we set $m = 0$. In our example in Fig. 2, the sequence (4) for $n = 5$ is 5, 2 ($m = 4$) and for $n = 9$ it is 9, 4, 1, 0 ($m = 0$).

The steps 3–7 in the algorithm ACR_FACTOR_ORACLE(w) will create the node $n+1$ and edges $n_{i_\ell} \xrightarrow{a} n+1$, $0 \leq \ell \leq k-1$ and then, at step 9, $S(n+1)$ gets the value m.

On the other hand, the inductive hypothesis says that in $\mathsf{Trie}(w[1..n])$ there is a path of suffix links starting from the deepest node such that the time values of the nodes on this path look like this:

$$n_0, \ldots, n_0, n_1, \ldots, n_1, n_2, \ldots, n_2, \ldots, n_{i_k}, \ldots, n_{i_k} \ , \tag{5}$$

where the last n_{i_k} corresponds to the shallowest node with that time value. It will have an a-son or else it is the root. Each sequence of $n_i, n_i, \ldots, n_i$ could actually contain a single n_i. We have also $S'(n_i) = n_{i+1}$, for all $0 \leq i \leq i_k - 1$.[a] In our example in Fig. 1, the sequence (5) for $n = 5$ is 5, 5, 5, 2, 2 corresponding to the nodes $\mathsf{baaba}_\mathbf{5}$, $\mathsf{aaba}_\mathbf{5}$, $\mathsf{aba}_\mathbf{5}$, $\mathsf{ba}_\mathbf{2}$, $\mathsf{a}_\mathbf{2}$; the node a has a b-son ($w[4] = \mathsf{b}$) and $\mathsf{time}(\mathsf{ab}) = 4 = m$; for $n = 9$ we have the sequence 9, 9, 9, 9, 9, 9, 6, 4, 1, 0 corresponding to the nodes $\mathsf{baababbab}_\mathbf{9}$, $\mathsf{aababbab}_\mathbf{9}$, $\mathsf{ababbab}_\mathbf{9}$, $\mathsf{babbab}_\mathbf{9}$, $\mathsf{abbab}_\mathbf{9}$, $\mathsf{bbab}_\mathbf{9}$, $\mathsf{bab}_\mathbf{6}$, $\mathsf{ab}_\mathbf{4}$, $\mathsf{b}_\mathbf{1}$, $\varepsilon_\mathbf{0}$; no node has a c-son. Notice the value 6 which does not appear in (4). Also, there are other nodes with time values 6 and 4 which do not appear on this path.

The steps 5–7 in the algorithm UKKONEN_TRIE(w) will create a-sons with time value $n+1$ for all nodes corresponding to the time values in (5) which do not have one. For each n_{i_ℓ}, the nodes with the time value n_{i_ℓ} will be merged in $\mathsf{Trie}(w[1..n+1])|_{\mathsf{time}}$ into one, so we obtain the same edges as those created in $\mathsf{Factor\_Oracle}(w[1..n+1])$. For each n_i which is not among the n_{i_ℓ}s, the edges added will be removed by (ii) in the statement. This is the case for $n = 9$ in our example: the edge $6 \xrightarrow{\mathsf{c}} 10$ will be removed. Lastly, if there are any n_{i_k}s in (5) without

[a]As the reader may have noticed, there is a relation between i_ℓs and the exponents e_is in the statement: $S'^{e_{n_{i_\ell}}}(n_{i_\ell}) = n_{i_{\ell+1}}$, which means $i_\ell = e_{n_{i_0}} + e_{n_{i_1}} + \cdots + e_{n_{i_{\ell-1}}}$ but we don't need these exact values of i_ℓs.

a-sons, the edges they get are removed when constructing $\mathsf{Trie\_Oracle}(w[1..n+1])$ from $\mathsf{Trie}(w[1..n+1])|_{\mathsf{time}}$. This is the case in our example for $n = 5$: the edge $2 \xrightarrow{\mathsf{b}} 6$ is removed. In all cases $\mathsf{Factor\_Oracle}(w[1..n+1])$ is obtained as claimed by (ii).

About (i), if $m = 0$, then $S'(n+1) = 0 = S(n+1)$. Assume $m \neq 0$ and denote the node corresponding to the last n_{i_k} in (5) by u. We have then, by Remark 2 above, that $ua \in \mathsf{suf}^*(w[1..n+1])$ which gives $m = \mathsf{time}(ua) \in S'^*(\mathsf{time}(w[1..n+1])) = S'^*(n+1)$. In our example, for $n = 9$, we have $S'(9) = 6$ but $S'^2(9) = 4 = S(9)$. □

7. Oracles from the suffix and the factor automata

Two other oracles for $\mathsf{Fact}(w)$ can be obtained naturally from the suffix and the factor automata by merging the states that were split during construction. We recall briefly the definition and the construction of the suffix automaton. For more details we refer to [7].

The suffix automaton of a string w, denoted $\mathsf{SA}(w)$, is the minimal deterministic automaton (not necessarily complete) that recognizes the set of suffixes of w, that is, $\mathsf{Suff}(w)$. Its states are therefore classes of the right syntactic congruence $\equiv_{\mathsf{Suff}(w)}$. The suffix automaton was introduced in [4].

In the (on-line) construction algorithm of $\mathsf{SA}(w)$, it is important to see how the syntactic congruence changes from a prefix x of w to the next one, say xa, where a is a letter. Let z be the longest suffix of xa that occurs in x and let z' be the longest string in the same $\equiv_{\mathsf{Suff}(x)}$-class as z. The classes of $\equiv_{\mathsf{Suff}(xa)}$ are the classes of $\equiv_{\mathsf{Suff}(x)}$ with two modifications:

- only when $z \neq z'$, the $\equiv_{\mathsf{Suff}(x)}$-class of z and z' is split into two $\equiv_{\mathsf{Suff}(xa)}$-classes: one containing the strings of the same length as z or shorter, the other containing the remaining ones (including z');
- a new class is added, that is, the $\equiv_{\mathsf{Suff}(xa)}$-class of the whole new string xa.

Define the *suffix function* $s_w : \mathsf{Fact}(w) \longrightarrow \mathsf{Fact}(w)$ by $s_w(v) =$ the longest $u \in \mathsf{Suff}(v)$ such that $u \not\equiv_{\mathsf{Suff}(w)} v$. The function s_w introduces the *s-links*[b] among the states of the suffix automaton. The suffix automaton for our running example is shown in Fig. 5 where the dotted lines are the s-links.

In terms of the automaton, the splitting of the syntactic congruence classes explained above translates into splitting of states. Preserving the above notation, when a new state is added for the prefix xa, then we have $s_w(xa) = z$ and if $z \neq z'$, then the state recognizing both z and z' in $\mathsf{SA}(x)$ needs to be split in $\mathsf{SA}(xa)$ into one state recognizing z and the shorter strings and another recognizing the ones longer than z, including z'. This is the case, for instance, in Fig. 5, when state number 9, corresponding to the prefix $xa =$ baababbab (of length 9) is added. In

[b]Our suffix function s is called suffix link in [7] but we already used that term with a different meaning.

this case we have $z = \mathsf{bab}$ is the longest suffix of xa that appears also in x, but $z' = \mathsf{baabab}$. Therefore, state 6 is split into 6 and 6′.

An oracle for $\mathsf{Fact}(w)$ can be obtained from the suffix automaton by merging together the states that were split during the algorithm constructing the suffix automaton and then making the result deterministic, as we did with $\mathsf{Trie}(w)|_{\mathsf{time}}$; denote the new oracle $\mathsf{SA\_Oracle}(w)$. We prove that $\mathsf{SA\_Oracle}(w)$ is isomorphic with $\mathsf{Factor\_Oracle}(w)$. What we actually show is that the result of merging the states of $\mathsf{SA}(w)$ is isomorphic with $\mathsf{Trie}(w)|_{\mathsf{time}}$.

An equivalence can be defined in an obvious way from the splitting of states. There is no need to be very formal in that respect. Assume the states in the same class are denoted $i, i', i'', \ldots, i^{(j)}, \ldots$. Because $\mathsf{SA}(w)$ can be obtained from $\mathsf{Trie}(w)$ by merging together isomorphic subtrees, it is enough to prove that all strings recognized at state $i^{(j)}$ by $\mathsf{SA}(w)$ have time equal to i. Assume xa is the prefix of length i of w. When the state i is added to the automaton, the strings recognized at i are precisely the suffixes of xa that did not appear so far (as factors of x). That means, suffixes strictly longer than z. But then i is also the first (leftmost) position where all these strings occur in w, that is, it is exactly the time at which they are considered by Ukkonen's algorithm. Later on, if the state i is split, the strings accepted at i are simply distributed among various $i^{(j)}$-states, but they all preserve the former $\mathsf{time} = i$.

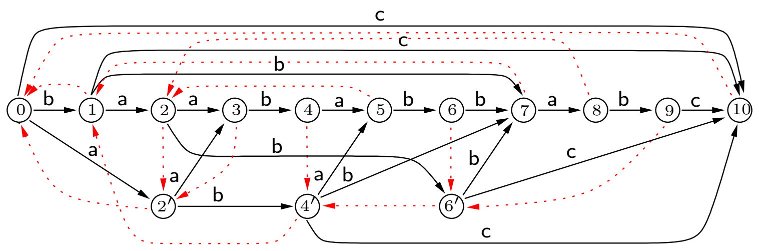

Fig. 5. $\mathsf{SA}(\mathsf{baababbabc})$

The second oracle for $\mathsf{Fact}(w)$ is obtained in the same way from the factor automaton of w, that is, the minimal deterministic automaton the recognizes $\mathsf{Fact}(w)$. The factor automaton was introduced in [6]. Its construction is similar to the one of $\mathsf{SA}(w)$ and the oracle is again obtained by merging the split states plus determinization. However, the proof that the same oracle is obtained is completely similar. For our running example, $\mathsf{baababbabc}$, the two machines are identical; the last letter c does not appear anywhere else in the string, making the two syntactic congruences $\equiv_{\mathsf{Suff}(w)}$ and $\equiv_{\mathsf{Fact}(w)}$ equal. If this last introduced oracle is denoted $\mathsf{FA\_Oracle}(w)$, then we proved

Theorem 9. *For any string w, the oracles* $\mathsf{Trie\_Oracle}(w)$, $\mathsf{SA\_Oracle}(w)$ *and*

$\mathsf{FA\_Oracle}(w)$ *are isomorphic.*

8. The factor oracle from the trie

We describe two ways of obtaining the factor oracle directly from the trie; different from constructing the trie oracle and then performing the eliminations in Theorem 8(ii). Both constructions here are similar to the one presented in Section 6 but they are independent of the factor oracle; in particular, they do not make reference to the S-links.

The first one uses the idea that the edges which the trie oracle has but the factor oracle does not appear from the fact that we may add an a-son to a node v even when there is already a node with the same time value as v which has an a-son. In our example, this happens when adding the edge $\mathsf{ba}_\mathbf{2} \xrightarrow{\mathsf{b}} \mathsf{bab}_\mathbf{6}$ when there is already $\mathsf{a}_\mathbf{2} \xrightarrow{\mathsf{b}} \mathsf{ab}_\mathbf{4}$ (subscripts show the time). Modifying Ukkonen's algorithm for tries so that such edges are not added would produce a graph whose quotient with respect to the time values produces precisely the factor oracle. For our example this new graph would look like the trie in Fig. 1 with the subtree rooted at bab removed. The new S'-links would be the same as the S-links in the factor oracle. However, the new suffix links would not be true suffix links but powers of those. Precisely, the new suffix link for a node u would link to the longest suffix of u that exists in the new graph. For instance, in our example, we would have the new (pseudo)suffix link of bbab pointing to $\mathsf{ab} = \mathsf{suf}^2(\mathsf{bbab})$ as the suffix bab is no longer in the new graph.

A better way to modify Ukkonen's algorithm is to add an a-son to a node v even if there exists another, say u, with the same time value and that has an a-son but the (new) time value of v will not be the index of the current letter in w but $\mathsf{time}(ua)$. This new version of the algorithm, say UKKONEN_TRIE_2(w), would have line 8 replaced by the following pseudocode (for clarity, denote the new time values by $\mathsf{time\_2}$ and the new S'-links by S''):

```
8.      $\mathsf{time\_2}(x) \leftarrow i$
8.1.    **if** $\mathsf{suf}^k(v) \neq \mathsf{nil}$ **then**
8.2.        $t \leftarrow \mathsf{time\_2}(\mathsf{suf}^k(v))$; $t' \leftarrow \mathsf{time\_2}(\mathsf{suf}^k(v)w[i])$; $\ell \leftarrow k-1$
8.3.        **while** $\mathsf{time\_2}(\mathsf{suf}^\ell(v)) = t$ **do**
8.4.            $\mathsf{time\_2}(\mathsf{suf}^\ell(v)w[i]) \leftarrow t'$; $\ell \leftarrow \ell - 1$
```

Denote the graph constructed by this algorithm $\mathsf{Trie\_2}(w)$ and let $\mathsf{Trie\_Oracle\_2}(w)$ be the quotient $\mathsf{Trie\_2}(w)|_{\mathsf{time\_2}}$. From the construction it should be clear that a deterministic automaton is obtained.

Remarks 1 and 2 hold as well for $\mathsf{Trie\_2}(w)$ with one exception. Not all nodes with a given $\mathsf{time\_2}$ are found on the suffix path from the deepest node with that $\mathsf{time\_2}$ value. The nodes with the same $\mathsf{time\_2}$ form a tree with the shallowest node as root. In our example, this happens for the nodes with $\mathsf{time\_2}$ value 4; the root is $\mathsf{ab}_\mathbf{4}$.

The results in Lemma 3, Corollaries 4 and 6 and Proposition 7 hold also for Trie_Oracle_2(w). The proof of the next lemma is similar to the one of Lemma 3.

Lemma 10. *In* Trie_Oracle_2(w)*, if $j \in S''^{*}(i)$, then $L(i) \subseteq L(j)$.*

Lemma 11. *For any w, we have:*
(i) Trie_Oracle_2(w) *recognizes at least all factors of w;*
(ii) the set of strings recognized by Trie_Oracle_2(w) *is closed under taking factors.*

Proof. For (i) there is nothing to prove as Trie_Oracle_2(w) is obtained by merging Trie_2(w). Lemma 10 implies (ii). □

Proposition 12. *For any w,* Trie_Oracle_2(w) *is an oracle for* Fact(w).

Proof. We have that Trie_Oracle_2(w) fulfils all five conditions in the definition of an oracle: (o_1) comes from (i) of Lemma 11, (o_2), (o_3), and (o_5) are clear, and (o_4) follows from Lemma 5. □

For our string baababbabc, the differences between Trie_2 and Trie are: (i) time_2(bab) $= 4 \neq 6 =$ time(bab) and (ii) $S''(9) = 4 \neq 6 = S'(9)$. Therefore, the two graphs Trie_Oracle_2(baababbabc) and Factor_Oracle(baababbabc) are identical. The next theorem says that this is always the case.

Theorem 13. *For any string w, we have*

(i) for any i, $S''(i) = S(i)$;
(ii) Factor_Oracle(w) *and* Trie_Oracle_2(w) *are the same.*

Proof. The proof is similar to the one of Theorem 8 but simpler. We show the two assertions to be true for each prefix of length n of w by induction on n. They hold for $n = 1$. Let us assume they are true for n and prove them for $n + 1$. Denote $w[n + 1] = a$. The path (4) of S-links from n to the first node with an a-son is here

$$n_0 = n, n_1 = S(n), n_2 = S^2(n), \ldots, n_k = S^k(n) \ . \tag{6}$$

Denote by m the a-son of n_k; if there is no node with an a-son on this path, then we set $m = 0$.

The steps 3–7 in the algorithm ACR_FACTOR_ORACLE(w) will create $n_\ell \xrightarrow{a} n + 1$, $0 \leq \ell \leq k - 1$ and $S(n + 1) = m$.

The corresponding path (5) of time_2 values starting from the deepest node in Trie_2($w[1..n]$) is

$$n_0, \ldots, n_0, n_1, \ldots, n_1, n_2, \ldots, n_2, \ldots, n_k, \ldots, n_k \ . \tag{7}$$

The last n_k corresponds to the shallowest node with that time value. It will have an a-son or else it is the root. Also, $S''(n_i) = n_{i+1}$, for all $0 \leq i \leq n_{k-1}$.

The steps 5–7 in the algorithm UKKONEN_TRIE_2(w) will create a-sons for all nodes corresponding to the time values in (5) which do not have one. Moreover, if there is more than one n_k in (7), then all the a-sons of such nodes will have time_2 value m. This can happen only when $m \neq 0$.

It is clear that Trie_2$(w[1..n+1])|_{\mathsf{time_2}}$ produces Factor_Oracle$[1..n+1])$ and so (ii) is proved.

For (i), we have $S''(n+1) = m = S(n+1)$. □

A direct construction of the factor oracle from the trie has been shown in [5]. However the algorithm presented there is less eficient than our method.

9. Properties of the factor oracle

The properties of the factor oracle from [1, 2] can be very easily deduced using Theorem 13. In Proposition 12 we proved that Factor_Oracle(w) is an oracle for the set of factors of w. Also Lemma 11 shows that the set of strings recognized by Factor_Oracle(w) is closed under taking factors.

Denote, for u factor of w, poccur$(u, w) = \min\{|z| \mid z = xu, w = zy\}$.

Lemma 14. *The shortest string recognized by* Factor_Oracle(w) *in i is a factor of w and is unique.*

Proof. This is the string labelling the shallowest (closest to root) node with time_2 value i in Trie_2(w). □

Denote the string in the above lemma by $\min(i)$. The next property is also clear.

Lemma 15. poccur$(\min(i), w) = i$.

Proof. We know that poccur$(\min(i), w) =$ time_2$(min(i)) = i$. □

Lemma 16. $\min(i)$ *is a suffix of any string recognized by* Factor_Oracle(w) *in i.*

Proof. The shallowest node with time_2 value i, corresponding to $\min(i)$, is on the suffix path of any node with time_2 i. □

Lemma 17. *Any factor u of w is recognized by* Factor_Oracle(w) *in $j \leq$* poccur(u, w).

Proof. We have in Trie_2(w), time_2$(u) \leq$ poccur(u, w). □

Lemma 18. *Any path in* Factor_Oracle(w) *whose label ends with* $\min(i)$ *leads to $j \geq i$.*

Proof. Such a path in Trie_2(w) leads to a node v such that the shallowest node of time_2 value i is on v's suffix path. It follows that $j =$ time_2$(v) \geq i$. □

Lemma 19. *Let v be a factor of w recognized by* $\mathsf{Factor\_Oracle}(w)$ *in i. Then any suffix of v is recognized in $j \le i$.*

Proof. The suffix of v is in $\mathsf{Trie\_2}(w)$ on the suffix path of v and therefore has a lower, or equal, $\mathsf{time\_2}$ value. $\square$

Lemma 20. $\mathsf{Factor\_Oracle}(w)$ *has at most* $2|w| - 1$ *edges.*

Proof. This follows from Lemma 5. $\square$

Denote $\mathsf{repet}(w, i)$ the longest suffix of $w[1..i]$ that appears at least twice in $w[1..i]$.

Lemma 21. *The reading of* $\mathsf{repet}(w, i)$ *in* $\mathsf{Factor\_Oracle}(w)$ *ends in* $S(i)$.

Proof. Starting from the deepest node of $\mathsf{Trie\_2}(w)$ with $\mathsf{time\_2}$ value i; it corresponds to the node $w[1..i]$. Moving up its suffix path, we encounter shorter suffixes until one which is already in $\mathsf{Trie\_2}(w)$ is found. That node has $\mathsf{time\_2}$ value $S(i)$ and its label is precisely $\mathsf{repet}(w, i)$. $\square$

10. Discussion and open problems

Our framework for discussing oracles for the set of factors of a string is provided by the following result.

Proposition 22. *Any oracle for the set of factors of a string w contains a quotient of* $\mathsf{Trie}(w)$ *as a subgraph.*

Proof. Let O be an oracle and define a relation on the nodes of $\mathsf{Trie}(w)$ by

$$u \equiv_O v \quad \text{iff} \quad \text{the paths labeled } u \text{ and } v \text{ in } O \text{ end in the same node .}$$

We claim that O must contain as a subgraph the quotient $\mathsf{Trie}(w)|_{\equiv_O}$.

First, it is clear that $\equiv_O$ is an equivalence relation with $|w| + 1$ classes. No two prefixes of w are in the same $\equiv_O$-class and denote the $\equiv_O$-class of the prefix of length i by i. Therefore, $\mathsf{Trie}(w)|_{\equiv_O}$ has the nodes $0, 1, \ldots, n$.

Consider an edge $i \xrightarrow{a} j$ of $\mathsf{Trie}(w)|_{\equiv_O}$. There must be then an edge $u \xrightarrow{a} ua$ in $\mathsf{Trie}(w)$ such that $u \equiv_O w[1..i]$ and $ua \equiv_O w[1..j]$. Therefore, reading u in O from 0 leads to i whereas reading ua from 0 leads to j. As O is deterministic, there must be an edge $i \xrightarrow{a} j$ in O, proving the claim. $\square$

Remark 23. Notice that, in particular, for any oracle O, $\mathsf{Trie}(w)|_{\equiv_O}$ is deterministic. Also, all edges in O which are not in $\mathsf{Trie}(w)|_{\equiv_O}$ can be eliminated and it still remains an oracle. Therefore, comparing oracles for the set of factors reduces to comparing quotients of the trie.

There exist particular examples, such as the string abcacdace from [5], in which the factor oracle is not the best possible oracle. The smaller oracle for the string abcacdace has one transition less than the factor oracle. However it accepts 16 errors compared to 13 errors accepted by the factor oracle. There are strings, such as abcacdaceacf for which there exists a smaller oracle which makes also less errors; in this case 33 errors whereas the factor oracle makes 39; see Figs. 6 and 7.

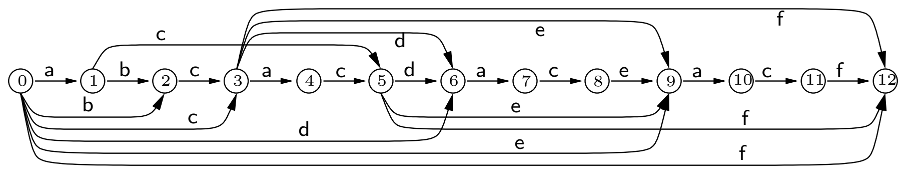

Fig. 6. Factor_Oracle(abcacdaceacf)

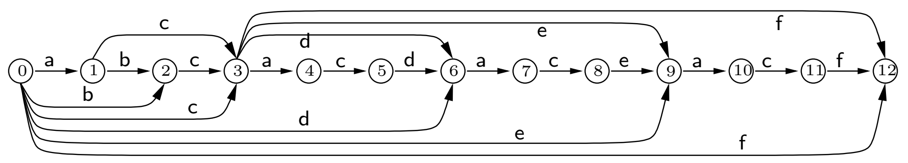

Fig. 7. A better oracle for the string abcacdaceacf.

Finally, we recall briefly the most important open problems about the factor oracle; they apply to all oracles for the set of factors of a string:

(1) Are factor oracle based string matching algorithms optimal on average, as conjectured by [1, 2]?
(2) What is the number of errors (maximal and average), that is, nonfactors that are recognized by the factor oracle? Examples were given in [13] where this number is exponential but they are over an unbounded alphabet. There are examples over a binary alphabet where it is still superpolynomial: for the string $\mathsf{abab}^2\mathsf{ab}^3\ldots\mathsf{ab}^n$, the number of errors is greater than $n!$.
(3) What is the average number of external transitions for the factor oracle or, put otherwise, what is its average size?
(4) Is there a simple strategy for building better oracles? Our framework of quotients of the trie allows comparing them.
(5) Characterize the set recognized by the factor oracle; not in terms of the factor oracle itself, like in [13].

References

[1] C. Allauzen, M. Crochemore, and M. Raffinot, Factor oracle: a new structure for pattern matching, *SOFSEM'99: theory and practice of informatics (Milovy)*, Lecture Notes in Comput. Sci. **1725**, Springer, Berlin, 1999, 295 – 310.

[2] C. Allauzen, M. Crochemore, and M. Raffinot, Efficient experimental string matching by weak factor recognition, *Combinatorial Pattern Matching (Jerusalem, 2001)*, Lecture Notes in Comput. Sci. **2089**, Springer, Berlin, 2001, 51 – 72.

[3] G. Assayag and S. Dubnov, Using factor oracles for machine improvisation, *Soft Comput.* **8**(9) (2004) 604 – 610.

[4] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas, The smallest automaton recignizing the subwords of a text, *Theoret. Comput. Sci.* **40**(1) (1985) 31 – 55.

[5] L. Cleophas, G. Zwaan, and B. Watson, Constructing factor oracles, *J. Autom. Lang. Comb.*, to appear.

[6] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45**(1) (1986) 63 – 86.

[7] M. Crochemore and C. Hancart, Automata for matching patterns, in G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages, Vol. II*, Springer-Verlag, Berlin, Heidelberg, 1997, 399 – 461.

[8] M. Crochemore, L. Ilie, and E. Seid-Hilmi, Factor oracles, in: O. Ibarra and H.-C. Yen, eds., *Proc. of CIAA'06*, Lecture Notes in Comput. Sci. **4094**, Springer, Berlin, Heidelberg, 2006, 78 – 89.

[9] R. Kato and O. Watanabe, Substring search and repeat search using factor oracles, *Inf. Process. Lett.* **93**(6) (2005) 269 – 274.

[10] A. Lefebvre and T. Lecroq, Compror: On-line lossless data compression with a factor oracle, *Inf. Process. Lett.* **83**(1) (2002) 1 – 6.

[11] A. Lefebvre and T. Lecroq, A heuristic for computing repeats with a factor oracle: application to biological sequences, *Int. J. Comput. Math.* **79**(12) (2002) 1303 – 1315.

[12] A. Lefebvre, T. Lecroq, and J. Alexandre, An improved algorithm for finding longest repeats with a modified factor oracle, *J. Autom. Lang. Comb.* **8**(4) (2003) 647 – 657.

[13] A. Mancheron and C. Moan, Combinatorial characterization of the language recognized by factor and suffix oracles, *Int. J. Found. Comput. Sci.* **16**(6) (2005) 1179 – 1191.

[14] E. Ukkonen, Constructing suffix trees on-line in linear time, *Proc. Information Processing'92, Vol. 1, IFIP Transactions A-12*, Elsevier, 1992, 484 – 492.