In-place Update of Suffix Array while Recoding Words

Matthias Gallé, Pierre Peterlongo, and François Coste

IRISA / INRIA Rennes Bretagne Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France
{matthias.galle, pierre.peterlongo, francois.coste}@irisa.fr

Abstract. Motivated by grammatical inference and data compression applications, we propose an algorithm to update a suffix array after the substitution, in the indexed text, of some occurrences of a given word by a new character. Compared to other published index update methods, the problem addressed here may require the modification of a large number of distinct positions over the original text. The proposed algorithm uses the specific internal order of suffix arrays in order to update simultaneously groups of entries, and ensures that only entries to be modified are visited. Experiments confirm a significant execution time speed-up compared to the construction of suffix array from scratch at each step of the application.

Keywords: suffix array, in-place update, dynamic indexing, word-interval

1 Motivation

In this paper, we propose an algorithm to update efficiently a suffix array, after substituting a word by a new character in the indexed text. This work is motivated by grammatical inference or grammar-based compression, along the lines initiated by SEQUITUR [21] in the framework formalized by Kieffer and Yang [11,12]. The goal is to infer a grammar G which generates only a given (long) sequence s in order to discover the structure that underlies the sequence, or simply, to compress the sequence thanks to a code based on the grammar. Learning and compression are often subtly intertwined (as for instance in the Occam's razor principle): in both cases the grammar is expected to be as small as possible. Kieffer and Yang introduced the definition of irreducible grammars and proposed several reduction rules allowing to transform a reducible grammar into an irreducible one, giving rise to efficient universal compression algorithms [11]. The sketch of these algorithms is to begin with a unique $S \rightarrow s$ rule generating the whole given sequence and essentially, to reduce iteratively the size of the grammar at each step by: 1) choosing a repeated pattern, 2) replacing the occurrences of the repeat by a new (non-terminal) symbol and 3) adding a new rewriting rule from this new symbol into the repeated pattern. For instance, the grammar $S \rightarrow uRvRw$, where u, v, w and R are substrings, and the length of R is strictly bigger than one, can be reduced at the first step into the grammar with two rules $S \rightarrow uAvAw$ and $A \rightarrow R$, where A is a new non-terminal symbol. At the following step, another repeated pattern, including eventually the new inserted symbol A, is selected and factorized by the introduction of a third rule, and so forth for the next steps. As a result, the algorithm returns a compact grammar which can be used to get a hierarchical point of view on the structure of the sequence or which can be encoded in order to get a better compression than by encoding directly the sequence.

Matthias Gallé, Pierre Peterlongo, François Coste: In-place Update of Suffix Array while Recoding Words, pp. 54–67.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

Algorithms of this kind are thus mainly based on the successive detection of repeats. They differ mostly in the order in which repeats are factorized. In SEQUITUR [21] and its variant [12], each repeat is replaced as soon as it is detected by a left to right scan of the sequence. More elaborate strategies for choosing the repeat to replace have been proposed. Kieffer and Yang proposed to replace longest matching substring [11]. Apostolico and Lonardi [3] proposed in their algorithm OFF-LINE to choose the substring yielding the best compression in a steepest-descent fashion. Efficient implementation of an elaborate choice of repeat often requires to use data structures from the suffix tree family. These index structures are well suited for efficient computations on repeats but they have to be built at initialization, and then updated at each step of the algorithm with respect to sequence modifications. Yet, as pointed out by Apostolico and Lonardi, most of the published work on updating a suffix tree, or – more general – on dynamic indexing problem – [20,7,17,8,6,23,5] focuses on localized modifications of the string and does not seem appropriate for replacing efficiently *more than one* occurrence of a given substring. Thus, index structures have usually to be built from scratch at each step of the algorithm. To our knowledge, only GTAC [16], an algorithm applied successfully on genomic sequences by Lanctot, Li and Yang, updates a suffix tree data structure after the deletion of all occurrences of a word. However, its updating scheme is specific to the longest matching substrings and seems difficult to adapt to other strategies.

In this paper, we propose a solution to the problem of updating efficiently an index structure while replacing some non-overlapping occurrences of a word of the indexed text by a new symbol. The first originality of our approach relies on the use of enhanced suffix arrays instead of suffix trees. Enhanced suffix arrays are known to be equivalent to suffix trees while being more space efficient [1]. They can be built in linear time [10,13,15] but non-linear algorithms [18,19] are usually more efficient for practical applications. A simple way of updating suffix array (instead of enhanced suffix array, thus without the same efficiency objective) by lazy bubble sort has been used in [22]. We propose here, to take advantage of the internal order offered by enhanced suffix arrays, to handle simultaneously groups of entries. This enables us to implement efficiently an update procedure for grammatical inference or grammar-based compression algorithm, choosing at each step a repeated substring, and replacing some or all of its occurrences by a new symbol.

2 Algorithm

2.1 Definitions and notations

A *sequence* is a concatenation of zero or more characters from an alphabet Σ. The number of characters in Σ is denoted by $|\Sigma|$. A sequence s of length n on Σ is represented by $s[0]s[1]\cdots s[n-1]$, where $s[i] \in \Sigma \ \forall 0 \leq i < n$. We denote by $s[i,j](j \geq i)$ the sequence $s[i]s[i+1]\cdots s[j]$ of s (if $j < i$ then $s[i,j] = \epsilon$, the empty string). In this case, we say that the sequence $s[i,j]$ occurs at position i in s. Its length, denoted by $|s[i,j]|$, is equal to $j-i+1$. Furthermore, the sequence $s[0,j]$ $(0 \leq j < n)$, also denoted by $s[..j]$, is called a prefix of s, and symmetrically, $s[i,n-1]$ $(0 \leq i < n)$, also denoted by $s[i..]$, is called a suffix of s.

Definition 1 (Suffix Array). *Consider a sequence s of length n over an alphabet Σ with a lexicographic order $\prec$ extensible to Σ^*. Let $\tilde{s} = s\$$, with a special character $\$$ not contained in Σ, lexicographically smaller than every element of Σ.*

The suffix array, denoted by sa, is a permutation of $[0..n]$ *such that:*

$$\forall\ i,\ 0 < i \leq n\ :\ \tilde{s}[sa[i-1]..] \prec \tilde{s}[sa[i]..]$$

Usually, the suffix array is used conjointly with an array called lcp, that gives the longest common prefix length between two suffixes whose starting positions are adjacent in sa. Formally,

$$lcp[0] = 0,$$
$$\textit{and } \forall\ i \in [1,n]\ :\ lcp[i] = k \textit{ such that}$$
$$\tilde{s}[sa[i-1]..][0,k-1] = \tilde{s}[sa[i]..][0,k-1] \textit{ and } \tilde{s}[sa[i-1]..][k] \neq \tilde{s}[sa[i]..][k].$$

Eventually, a third array called isa (for inverse suffix array) may be used conjointly with sa and lcp. This array gives, for a position p in s, the index i in sa such that $sa[i] = p$. *Thus* $sa[isa[p]] = p$.

The association of *sa*, *lcp* and *isa* arrays is called an *Enhanced Suffix Array* (*ESA*). An *ESA* enables $O(n)$ computation of occurrences of different kinds of repeats (repeats, maximal repeats [9,14] or super maximal repeats [1,9]).

In this paper, we propose to update an *ESA*, deleting and moving some of its indexes and keeping *lcp* consistent. In order to avoid shifting set of entries, we link consecutive entries using two additional arrays called *next* and *prev*. Thus, $next[i]$ (resp. $prev[i]$) gives the index of the next (resp. previous) valid entry in the *ESA*. Initially, $next[i] = i + 1$ and $prev[i + 1] = i$. We call the set *ESA* plus *next* and *prev* arrays the ESA_{DL} for *Double Linked Enhanced Suffix Array*.

It is worth noticing that an ESA_{DL} has not exactly the same properties as an *ESA*. Indeed, going from an entry i to entry $i + j$ may be done in constant time on an *ESA*, while this operation in an ESA_{DL} requires $O(j)$ time, as the *next* array has to be used j times.

Anyway, an ESA_{DL} still allows the detection of repeats (general repeats, maximal repeats or super maximal repeats) in linear time, because the algorithms used advance one by one over the arrays like most of the algorithm over *ESA* (a notable exception is the algorithm searching for a subsequence proposed in [25]).

We propose an *in-place* solution, where we always work with the same arrays and only update the values of their fields. Moreover, during the whole process, we modify only the *prev*, *next* and *lcp* arrays. Arrays *sa* and *isa* remain unchanged. This approach forces to extend the in-place behavior to the sequence: we also add two arrays to imitate a double linked list over the sequence.

The j^{th} position after position i, is denoted by $i \oplus j$. We compute $i \oplus j$ using links between sequence positions, indicating for each position its successor. Similarly $i \ominus j$ points to the j^{th} position before i. We define that, if $i \oplus j$ (respectively $i \ominus j$) is out of range, then $i \oplus j = n + 1$ (respectively $i \ominus j = -1$).

The left context tree. One of the most useful characteristic of a suffix array is that all indexes corresponding to suffixes starting with the same word correspond to an adjacent block. We define here the corresponding concept of word interval. Based on this, we will define the *left context tree* of a word ω where the nodes correspond to a left context of ω.

An ω-interval is the set $\{k\ :\ \exists \ell,\ k = isa[\ell] \wedge \tilde{s}[\ell..\ell + |\omega| - 1] = \omega\}$. This can also be denoted as an $[i..j]$-interval, where i and j are respectively the lowest and highest

indices of an ω-interval. Let us note that different words can share the same interval. More precisely, any pair of words ω and $\omega\alpha$ share the same interval if each occurrence of ω is followed by α.

This definition is thus slightly more general than the definition of ω-interval given by Abouelhoda, Kurtz and Ohlebusch [1], since we also define ω-interval for words leading to implicit nodes of a compact suffix tree, and not only to internal nodes.

The *left context tree of* ω ($\omega \in \Sigma^*$) for a sequence $\tilde{s}$ is an implicit tree whose nodes are v-intervals ($v \in \Sigma^*$) such that:

- the root is the ω-interval
- for each v-interval node corresponding to a non-empty interval, its children are all the av-intervals, for all $a \in \Sigma$
- the leaves are empty intervals

Given the isa array, it is easy to obtain the parent of a node. Let $[i..j]$ be an av-interval node. Given $k \in [i..j]$, $isa[sa[k]+1]$ is an index belonging to the v-interval. Inversely, $isa[sa[k]-1]$ belongs to one of the child interval. The exact child depends on the character at $\tilde{s}[sa[k]-1]$. We introduce the *successor* and *predecessor* notations:

$$successor(i) = \begin{cases} isa[sa[i] \oplus 1] & \text{if } sa[i] \oplus 1 \neq n+1 \\ n+1 & \text{otherwise,} \end{cases}$$

and

$$predecessor(i) = \begin{cases} isa[sa[i] \ominus 1] & \text{if } sa[i] \neq 0 \\ -1 & \text{otherwise.} \end{cases}$$

One may remark that $predecessor(i)$ is the equivalent of the "*suffix link*" in a suffix tree [26].

The problem that an *ESA* update algorithm must face is that the changes over the occurrences of a word ω not only affect the ω-interval, but also some of the $v\omega$-intervals ($v \in \Sigma^*$). The core of our algorithm is based on moving $v\omega$-interval in constant time, using the two following properties implied by the internal order of suffix arrays:

Proposition 2. *Let $[i..j]$ be an v-interval ($v \in \Sigma^*$), and $k_1, k_2 \in [i..j]$ with $k_1 > k_2$ and such that $predecessor(k_1)$ and $predecessor(k_2)$ belong to the same αv-interval ($\alpha \in \Sigma$). Then $predecessor(k_1) > predecessor(k_2)$.*

Proposition 3. *With $i < j$, the longest common prefix between $\tilde{s}[sa[i]..]$ and $\tilde{s}[sa[j]..]$ is $min_{k \in [next[i],j]}(lcp[k])$.*

In this paper, we consider that the grammatical inference or grammar based compression algorithm proceeds by steps. At each step, the alphabet grows because of the introduction of a new character: Σ_k will denote the alphabet in step k. In each, of this steps, the algorithm **i)** finds a repeat $\mathcal{R}_k$ in a sequence $\tilde{s}^{(k)}$ defined on the alphabet Σ_k and returns a list $\mathcal{O}_k$ of non-overlapping occurrences of $\mathcal{R}_k$ **ii)** updates the sequence $\tilde{s}^{(k)}$ and its associated ESA_{DL} replacing the given occurrences of $\mathcal{R}_k$ by a single new character $\mathcal{C}_k$, thus defining a new alphabet $\Sigma_{k+1} = \Sigma_k \cup \{\mathcal{C}_k\}$. The modified sequence is then called $\tilde{s}^{(k+1)}$. The whole iterative process stops either if no more repeat is found in the sequence or after a fixed number of iterations.

Our contribution focuses on updating the ESA_{DL}, at each step k of this algorithm (part **ii**).

In the next sections, we describe how to perform the three tasks needed for updating an ESA_{DL} at each step k: **1)** delete entries of suffixes starting inside an $\mathcal{R}_k$ occurrence; **2)** move entries with respect to the new alphabetic order; and **3)** update *lcp* array with respect to recoding occurrences of $\mathcal{R}_k$ by one single character. Note that a few values of the *lcp* array are also modified during part 1 and 2, but only as a consequence of deletions and moves.

2.2 Delete entries of suffixes occurring inside $\mathcal{R}_k$ substituted occurrences

entry	*lcp*	suffix
prev[*j*]	4	$ATAC\ldots$
j	2	$ATGA\ldots$
next[*j*]	3 **2**	$ATGT\ldots$

Figure 1. Deletion of entry j.

By replacing the word $\mathcal{R}_k$ by a single letter, the sequence is compressed and so is its ESA_{DL}: consequently, any suffix of sequence $\tilde{s}^{(k)}$ appearing inside an $\mathcal{R}_k$ substituted occurrence must be deleted. Thus for i in $\mathcal{O}_k$ and for ℓ in $[1, |\mathcal{R}_k| - 1]$, suffix $\tilde{s}^{(k)}[i \oplus \ell..]$ and the associated index in the suffix array $j = isa[i \oplus \ell]$ have to be removed. We simulated this deletion by *jumping over it* by setting *next* and *prev* arrays to their previous and next index: $next[prev[j]] \leftarrow next[j]$ and $prev[next[j]] \leftarrow prev[j]$. Furthermore, the *lcp* value of the index following j ($lcp[next[j]]$) has to be modified according to the deletion of index j. As a consequence of proposition 3, after the deletion of index j, the longest common prefix of entry $next[j]$ is equal to the minimal longest common prefix value of entries j and $next[j]$.

An example is shown in Figure 1 where the deletion of entry j affects the $lcp[next[j]]$ that now should contain the length of longest common prefix between $ATGT$ and $ATAC$ which is 2, equal to the longest common prefix of $ATGT$, $ATGA$ and $ATAC$.

Algorithm 1 presents the procedure for deleting indexes. The notation END refers to the last index of the suffix array ($prev[n + 1]$).

Algorithm 1 Delete entries at step k, replacing $\mathcal{R}_k$ by $\mathcal{C}_k$

delete_entries$\{ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k\}$

```
1: for i ∈ O_k do
2:    for ℓ ∈ [1, |R_k| − 1] do
3:       j ← isa[i ⊕ ℓ]
4:       if next[j] ≠ END then
5:          lcp[next[j]] ← min(lcp[j], lcp[next[j]])
6:       end if
7:       next[prev[j]] = next[j]
8:       prev[next[j]] = prev[j]
9:    end for
10: end for
```

2.3 Move entries, with respect to new alphabetic order

After replacing the word $\mathcal{R}_k$ by the new character $\mathcal{C}_k$, some ESA_{DL} lines may be misplaced with respect to the chosen order of $\mathcal{C}_k$ in Σ_{k+1}.

Entries in the $\mathcal{R}_k$-interval are potentially misplaced. Moreover, for $v \in \Sigma_k^*$, index entries inside an $v\mathcal{R}_k$-interval are misplaced if the substitution of $\mathcal{R}_k$ into $\mathcal{C}_k$ affects

their lexicographical order with respect to the previous and next index over the suffix array. Thus, lines belonging to node-intervals of the left-context tree of $\mathcal{R}_k$ may have to be moved.

In our approach, we decided to give to $\mathcal{C}_k$ the largest rank in the lexicographic order of the alphabet Σ_k, *i.e.* $\forall\, \alpha \in \Sigma_k \,:\, \alpha \prec \mathcal{C}_k$.

With respect to this arbitrary choice, the $\mathcal{R}_k$-interval is moved after the last entry of the suffix array. Furthermore, for any $v \in \Sigma_k^*$, the $v\mathcal{R}_k$-interval is moved after the last entry of the v-interval.

If an $v\mathcal{R}_k$-interval is already at the end of the v-interval (it is naturally well ordered), for any $v' \in \Sigma_k^*$, the $v'v\mathcal{R}_k$-interval is also at the end of the $v'v\mathcal{R}_k$-interval and has not to be moved.

Algorithm 2 Restore consistency of suffix array order

$update\_order\{ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{start}, depth, move\}$

if Couple $(i_{start}l, depth)$ already treated during another recursion call **then**
  End procedure
end if
$i \leftarrow i_{start}$
while $i \neq END \wedge lcp[next[i]] \geq depth + |\mathcal{R}_k|$ **do**
  $i \leftarrow next[i]$
end while
$i_{end} \leftarrow i$
$minLCP \leftarrow min_{j \in [i_{start}, i_{end}]} lcp[j]$
if *move* **then**
  while $i \neq END \wedge lcp[next[i]] \geq depth$ **do**
    $i \leftarrow next[i]$
  end while
end if
$i_{dest} \leftarrow i$
if $i_{end} \neq i_{dest}$ **then**
  $lcp[next[i_{end}]] \leftarrow min(lcp[next[i_{end}]], minLCP)$
  $lcp[i_{start}] \leftarrow depth$
  if $i_{start} = i_{first} \wedge depth \neq 0$ **then**
    $i_{first} \leftarrow next[i_{end}]$
  end if
  move_group$(i_{start}, i_{end}, i_{dest})$
else
  $lcp[i_{start}] \leftarrow min(lcp[i_{start}, depth)$
  $move \leftarrow false$
end if
$i \leftarrow i_{start}$
while $i \neq next[i_{end}]$ **do**
  $newdepth \leftarrow depth+$ (**if** $predecessor(i) \in \mathcal{O}_k$ **then** len **else** 1)
  if $move \vee (sa[prev[predecessor(i)]] > newdepth \wedge sa[prev[predecessor(i)]] \oplus newdepth \in \mathcal{O}_k)$ **then**
    $update\_order(ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, predecessor(i), newdepth, i_{dest} \neq i_{end})$
  end if
  $i \leftarrow next[i]$
end while

Based on this property, our algorithm uses a recursive approach in order to move groups. The recursion starts on the initial $\mathcal{R}_k$-interval. During recursion, if the group of an $v\mathcal{R}_k$-interval is moved, the recursion continues on groups of $\alpha v\mathcal{R}_k - intervals$, with $\alpha \in \Sigma_k$.

From a theoretical point of view, the algorithm starts on the root of the left-context tree of $\mathcal{R}_k$ and if the group corresponding to the interval of the node is moved, it recursively treats its children in a breadth first traversal (a FIFO is used).

In practice, the recursion on an $v\mathcal{R}_k$-interval works as follow:

1. detects the end position of the $v\mathcal{R}_k$-interval,
2. detects the end position of the v-interval,
3. if necessary:
   3.a. moves the group to the end position of the v-interval,
   3.b. call the recursion on predecessors of entries of the group.

During a call on predecessor of an entry of the group, either this is the first time the matched group is called and by construction the call is done on its first element, or the group was already treated, and the recursion stops.

The algorithm for this step is shown in algorithm 2. This recursion function receives three parameters besides the data structures: the starting position of the group, the current depth over the left-context and a boolean flag (see below).

At first, the end of the $v\mathcal{R}_k$-interval is found (lines 2, 2 and 2).

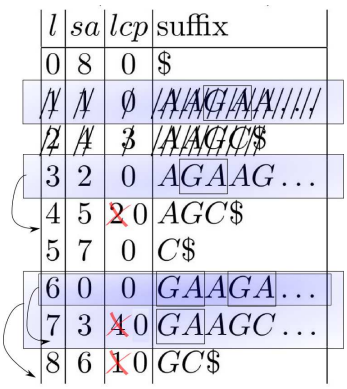

Figure 2. Moves induced by substituting GA by $\mathcal{C}_1$.

This is done from the first element of the interval, following the *next* array while the visited entry corresponds to a suffix starting with $v\mathcal{R}_k$ ($lcp \geq |v| + |\mathcal{R}_k|$). After finding the extremes of the group, the destination index of this group according to the chosen order for the new character is found (lines 2, 2 and 2). This is done by finding the end of the v-interval in the same way ($lcp \geq |v|$).

Moving now the group to its new position is simple and is done in constant time. Thanks to the well-ordered property of the suffix array, the whole interval is moved by changing only the delimiting positions. Let $i_{start}, i_{end}, i_{dest}$ be respectively the starting and ending positions of the $v\mathcal{R}_k$-interval, and the last position of the v-interval. Move the group $[i_{start}, i_{end}]$ to the position after i_{dest} is simply done by jumping over the group and *inserting* it into i_{dest} and $next[i_{dest}]$. See the algorithm 3 for implementation details.

Two longest common prefix values are modified as a consequence of the deletion of the group and its insertion:

1. $lcp[next[i_{end}]]$: contains the value of the length of the longest common prefix between $prev[i_{start}]$ and $next[i_{end}]$, which according to proposition 3, is the minimum of the lcp values of the group and itself
2. $lcp[i_{start}]$: we assign to it the value of $depth$, that is the correct value over $\tilde{s}_{k+1}$. This serves also to set a stop-point for future recursions calls (see below).

As i_{first} points to the first line over the suffix array that contains a selected repetition, we also update i_{first} (line 2) if this line is moved.

Figure 2 shows the ESA_{DL} of sequence $GAAGAAGC$, where $\mathcal{R}_1 = GA$ is substituted by $\mathcal{C}_1$. One remarks that the initial interval of suffixes starting with GA (indexes 6 and 7) is moved as well as suffix starting with AGA (index 3). Note also that suffix starting with $GAAGA$ has to be moved with respect to suffix $GAAGC$.

Algorithm 3 Move the group $[i_{start}, i_{end}]$ after the position i_{dest}

$move\_group\{ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{start}, i_{end}, i_{dest}\}$
1: $next[prev[i_{start}]] = next[i_{end}]$
2: $prev[next[i_{end}]] = prev[i_{start}]$
3: $next[i_{end}] = next[i_{dest}]$
4: $prev[next[i_{dest}]] = i_{end}$
5: $next[i_{dest}] = start$
6: $prev[i_{start}] = i_{dest}$

A special case Once an interval is treated, the recursion continues either if the current group was moved, or in the special case described in what follows.

Consider for instance the following case, where the substituted repeat is TA.

i C**TA**TT**TA**C...
i+1 C**TA**TT**TA**G...
i+2 C**TA**T**TA**...,

and suppose that the TTA-interval containing the index $isa[sa[i+2] \oplus 3]$ (the underlined suffix in the figure) was already at its right position and therefore has not to be moved. So, its children in the left-context tree are not considered for future moves, and as a consequence, neither is index $i+2$. Supposing that we cut the recursion here, that means that when treating the $CTATT$-interval, $lcp[i+2] = 5$. This interval ends at the index $i+1$, but because we use the lcp array to detect it, we also consider index $i+2$ as part of the $CTATT$-interval.

To resolve this special case, the recursion continues even when the current interval was not moved. In this case, it will never be necessary to move an interval, but maybe update some lcp values to set *stop-points* for future recursion calls.

This is the reason for introducing the last parameter in algorithm 2 (the boolean flag *move*). It differentiates the normal case (when it is necessary to detect the destination index and move the interval) from the case in which the current interval is considered only to set a *stop-point* at the first index of the interval. The recursion continues in both cases.

Filtering non substituted $\mathcal{R}_k$ occurrences Among each $v\mathcal{R}_k$-interval, suffixes starting with $v\mathcal{R}_k$ where $\mathcal{R}_k$ is not substituted (whose position does not belong to $\mathcal{O}_k$) may occur. The associated entries in the ESA_{DL} should not be moved with the $v\mathcal{R}_k$-interval. Thus, before to apply the recursive procedure previously exposed, a straightforward *filtering step* is applied. During the recursion, each line i of each group is first checked in order to detect if it corresponds to an entry of a selected occurrence ($sa[i] \oplus depth \in \mathcal{O}_k$). Once detected a non-selected occurrence, we move it to the beginning of the group (before i_{start}). As previously mentioned, this also involves modifications of the lcp array for maintaining its consistency.

2.4 Update lcp values after the substitution of $\mathcal{R}_k$ occurrences to a single character

The substitution of any occurrence of $\mathcal{R}_k$ of length $|\mathcal{R}_k| \geq 2$ by $\mathcal{C}_k$ of length 1 involves the modification of the length of all common prefixes involving such an occurrence.

In the previous step, it was easy to update the lcp values of the limits of the intervals while they were moved. In this step, we update the lcp values of the internal position of the intervals.

For this, we traverse the left-context tree of $\mathcal{R}_k$. Contrary to the moving step, where it was possible to move one line several times, in this step we update each *lcp* index only once. To do this, we recalculate all the *lcp* values for the root ($\mathcal{R}_k$-interval) and use this information to update the *lcp* of the other intervals.

As a consequence of propositions 2 and 3, the *lcp* between two indexes of the same interval-node is simply one plus the *lcp* between their successor indexes belonging to the parent interval-node:

Let i, j belong to the same *aw*-interval and let us assume that $i > j$.

Then $lcp(\tilde{s}[sa[i]..], \tilde{s}[sa[j]..]) = min_{\ell \in [next[successor(i)], successor(j)]} lcp[\ell]$

With this inductive approach, it is sufficient to re-calculate the *lcp* of only the first interval (the root of the left-context tree). This is straightforward (see algorithm 4).

Algorithm 4 Calculate the value of the *lcp* for index i

```
recalculate_lcp{ESA_DL, i}
 1: lcp[i] ← 0
 2: if prev[i] ≥ 0 then
 3:    i ← sa[i]
 4:    j ← sa[prev[i]]
 5:    while i < n ∧ j < n ∧ s[i] = s[j] do
 6:       i ← i ⊕ 1
 7:       j ← j ⊕ 1
 8:       lcp[i] ← lcp[i] + 1
 9:    end while
10: end if
```

During the iterative call, if an index already treated appears, it is skipped. Indeed, its *lcp* value is then up-to-date. The pseudo-code for this step is exposed in algorithm 5.

Algorithm 5 Update *lcp* of step k

```
update_lcp{ESA_DL^(k), R_k, O_k}
 1: q ← queue()
 2: for i ∈ O_k do
 3:    recalculate_lcp(ESA_DL^(k), isa[i])
 4:    q.push((predecessor(isa[i]), 1))
 5: end for
 6: while not q.empty() do
 7:    (i, depth) ← q.top
 8:    q.pop
 9:    if i ≥ 0 ∧ lcp[i] not already updated ∧ lcp[i] ≥ depth then
10:       lcp[i] ← (min_{j∈[next[successor(prev[i])], successor(i)]} lcp[j]) + 1
11:       q.push((predecessor(i), depth + 1))
12:    end if
13: end while
```

Because in each step we use the value of all the lines of the previous group, we traverse once again the left context tree in a breadth-first order.

3 Efficiency

The space complexity is in $O(n)$. The ESA_{DL} structure needs to complete the *ESA* with two arrays of length n. During the execution, a queue of length $O(n)$, plus an

array of length n are used to check in constant time whether a couple $(i, depth)$ was already used.

The worst case time complexity of the update algorithm is bounded by $O(n^2)$. This case is reached while replacing for instance AA occurrences in an ESA_{DL} indexing the text A^nT. A better bound on time complexity could be obtained by considering amortized complexity, but it will still be unlikely to be better than the $O(n)$ complexity required for building the suffix array from scratch. Nevertheless, the algorithms building suffix arrays that currently perform best in practical cases, are not the linear ones (see [24] for a complete description of the different suffix array construction algorithms and their strengths). We propose in this section to evaluate the practical efficiency of our algorithm.

A prototype implementing the proposed algorithm has been developed using the C++ language. It is available at `http://www.irisa.fr/symbiose/people/galle/update_sarray/`. It has been tested on different types of text. For the sake of briefness, in this paper we only report the results on the following classical corpora from the literature:

- the standard and large Canterbury corpus (`http://corpus.canterbury.ac.nz/`, [4]),
- the Purdue corpus (`http://www.cs.ucr.edu/~stelo/Offline/`, [2])

Similar tests on other corpora can be found on our internet site.

We compared the execution times of our algorithm with the linear time suffix array construction algorithm proposed by Kärkkäinen and Sanders [10], and non-linear algorithm of Larsson and Sadakane [18] that is in practice faster. Both source codes were retrieved from the Internet sites specified in the associated articles. Note that Kärkkäinen and Sanders' code "strives for conciseness rather than for speed" [10]. The Manzini and Ferragina's algorithm [19], doesn't fulfill our requirement of variable alphabet size, it was then not used for our experiments. The tests were executed on 1 GHz AMD Opteron processors with 4 GB of memory.

First, to have an idea of the complexity of the algorithm, we studied how the length of the sequence influences the execution time of the algorithm. From the large Calgary corpus, we extracted sequences of different length by considering successively bigger (by steps of 100 characters) prefixes of the sequences. On each extracted sequence, we performed 250 iterations of selecting a random repeat, replacing it over the sequence by a new character and updating the associated suffix array. Time (user + system time) required for updating the suffix array was reported, averaged over 5 different runs corresponding to 5 different random seeds. The same experiments, replacing the update algorithm by the "from scratch" construction algorithms of the suffix array by Kärkkäinen and Sanders (*K & S*) and Larsson and Sadakane (*L & S*) have been performed. The plots, shown in figure 3, confirm that the execution time of our updating algorithm is not directly correlated to the length of the sequence, and is significantly smaller than the execution time required by reconstruction "from scratch" algorithms, especially when the length of the sequence increases.

We present a more exhaustive evaluation and comparison on all the corpora using different strategies for the selection of the repeated word. In each test we performed 500 iterations of selecting a repeat, replacing it over the sequence and updating (or building from scratch) the associated suffix array. The different strategies for the selection of the repeat were:

- take a random one (using the same seed for the random number generator),
- take the longest,

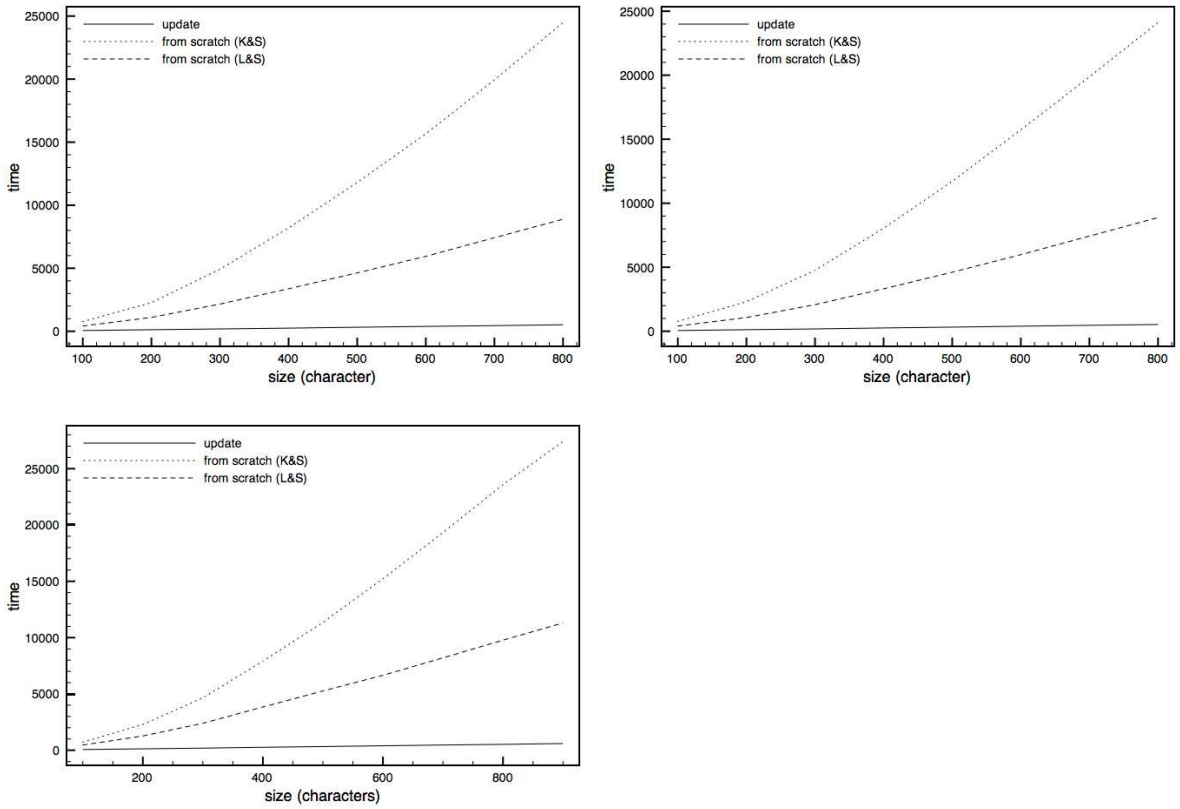

Figure 3. Large corpus: *bible.txt*, *world192.txt* and *E.coli*. Times are given in hundredth of seconds

– take the one that covers the maximal number of positions[1].

Results are given in figure 4 (page 67). For each selection strategy, we report time (user + system time) spent in updating ESA_{DL} with our algorithm (column *update*), and time spent in building *ESA* from scratch at each iteration with the linear algorithm from Kärkkäinen and Sanders (column *K & S*) and the algorithm from Larsson and Sadakane (column *L & S*). For easier comparison, the ratios of the time spent by each of the two "from scratch" algorithms over the update algorithm are also given.

Some of the files (*fields.c, grammar.lsp* and *xargs.1*) are too small to draw significant conclusions, but results are shown here for the sake of completeness. On the other files, results show that a significant speedup is usually achieved by using our algorithm. The main exceptions are the $Spor\_All\_2x.fasta$ files (an artificial file obtained by concatenating $Spor\_All.fasta$ with itself) from the Purdue corpus, and the *ptt5* file from the Canterbury corpus (a fax image with very long zones of the same byte). One can also remark that the ratio is less favorable when the repeat to replace is chosen according to the maximal compression strategy. On the one hand, in each iteration the resulting sequence is smaller and the suffix array creation from scratch for this sequence faster. On the other hand, there are more positions affected by the substitution and this affects the update algorithm.

[1] Maximisation of $(|\mathcal{O}_k| - 1) * (|w| - 1) - 1$, corresponding to a maximal compression approach.

These cases allow us to illustrate an intrinsic limit of the update approach when the length of the sequence is highly reduced by recoding: when the number of positions to update is larger than the number of positions in the resulting sequence, it may be worth adopting the "from scratch" construction algorithm (let us remark that the best algorithm to use can vary along the iterations). A solution to handle these extreme cases, would be to design a criterion on the repeat and its coverage to automatically choose the best algorithm to use (eventually at each iteration).

4 Conclusion and future work

We introduced in this paper an approach allowing to keep up-to-date an enhanced suffix array with respect to the substitution of some of the occurrences of a word in the indexed text We didn't consider singular insertions or deletions, but simultaneuos substitution.This is of particular interest for grammatical inference or grammar based compression methods which are using these data structures and are performing iteratively a large number of such substitutions.

Our approach uses the specific internal order of suffix arrays to update simultaneously groups of adjacent entries and ensures that only entries to be modified are visited. This specific property of the suffix arrays allows to design an efficient update procedure which has been implemented and tested on classical corpora. The experimentation confirms that, in regard to the direct method reconstructing the suffix array, our approach enables significant speed-up of the execution time.

However, in some cases, the update method is less efficient than building the enhanced suffix array from scratch. Intuitively, when the number of lines to change is larger than the number of lines in the new suffix array, a reconstruction algorithm is likely to be more efficient than an update approach. In order to be even more efficient, a criterion allowing to decide automatically which algorithm to use could be designed. This would require a finer complexity analysis of the update algorithm, but also of the chosen building algorithm, in order to identify easy-to-compute key parameters involved in the execution time complexity.

Of course, the question of the existence of a practical efficient $O(n)$ algorithm remains open. But the results on the construction of suffix arrays suggest that a better way of improvement could be the design of other practical update algorithms. Finally, these results have been obtained by using a suffix array. It would be interesting to study how easily this approach can be adapted to suffix trees and how much it depends on the suffix array specific properties.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Compression of biological sequences by greedy off-line textual substitution*, in Proc. DCC, 28-30 March 2000, pp. 143–152.
3. A. Apostolico and S. Lonardi: *Off-line compression by greedy textual substitution*, in Proc. IEEE, vol. 88, Nov. 2000, pp. 1733–1744.
4. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in Proc. Conference on Data Compression, Washington, DC, USA, 1997, IEEE Computer Society, p. 201.
5. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane: *Compressed indexes for dynamic text collections.* ACM Transactions on Algorithms, 3(2) May 2007.

6. P. Ferragina, R. Grossi, and M. Montangero: *On updating suffix tree labels.* Theor. Comp. Science, 201(1-2) 1998, pp. 249–262.
7. E. Fiala and D. H. Greene: *Data compression with finite windows.* Comm. ACM, 32(4) 1989, pp. 490–505.
8. M. Gu, M. Farach, and R. Beigel: *An efficient algorithm for dynamic text indexing*, in Proc. the ACM-SODA, 1994, pp. 697–704.
9. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Jan. 1997.
10. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. ICALP, Springer, 2003.
11. J. Kieffer and E.-H. Yang: *Grammar-based codes: A new class of universal lossless source codes.* IEEE TIT, 46 2000.
12. J. Kieffer and E.-H. Yang: *Grammar-based codes: a new class of universal lossless source codes.* IEEE TIT, 46 2000.
13. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. CPM, vol. 2676, 2003, pp. 200–210.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. IEEE FOCS, New York, USA, 1999, IEEE Computer Society Press, pp. 596–604.
15. K. D. Kyue, S. J. Seop, P. Heejin, and P. Kunsoo: *Linear time construction of suffix arrays*, in Proc. Combinatorial Pattern Matching, vol. 2676, 2003, pp. 186–2003.
16. J. K. Lanctot, M. Li, and E.-H. Yang: *Estimating dna sequence entropy*, in Proc. ACM-SODA, 2000, pp. 409–418.
17. N. J. Larsson: *Extended application of suffix trees to data compression*, in Proc. DCC, 1996, pp. 190–199.
18. N. J. Larsson and K. Sadakane: *Faster suffix sorting*, Tech. Rep. LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.
19. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm.* Algorithmica, 40(1) 2004, pp. 33–50.
20. E. M. McCreight: *A space-economical suffix tree construction algorithm.* J. ACM, 23(2) 1976, pp. 262–272.
21. C. Nevill-Manning and I. Witten: *Identifying hierarchical structure in sequences: A linear-time algorithm.* J. AI Research, 7 1997, pp. 67–82.
22. C. Nevill-Manning and I. Witten: *On-line and off-line heuristics for inferring hierarchies of repetitions in sequences.* Proc. IEEE, 88(11) Nov 2000, pp. 1745–1755.
23. S. C. Sahinalp and U. Vishkin: *Efficient approximate and dynamic matching of patterns using a labeling paradigm*, in FOCS, 1996.
24. K.-B. Schürmann and J. Stoye: *An incomplex algorithm for fast suffix array construction.* Software - Pract. and Exp., 37(3) 2007, pp. 309–329.
25. J. S. Sim: *Time and space efficient search for small alphabets with suffix arrays*, in Proc. FSKD, 2005.
26. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14 1995, pp. 249–260.

sequence	length (chars.)	random					maximal length					maximal compression				
		update	K & S	L & S	ratio K & S	ratio L & S	update	K & S	L & S	ratio K & S	ratio L & S	update	K & S	L & S	ratio K & S	ratio L & S
CANTERBURY CORPUS																
alice29.txt	152089	163	2812	1497	17.25	9.18	192	2357	1371	12.28	7.14	269	1091	510	4.06	1.90
asyoulik.txt	125179	131	2111	1109	16.11	8.47	127	1727	1059	13.60	8.34	182	866	405	4.76	2.23
cp.html	24603	15	132	95	8.80	6.33	15	96	64	6.40	4.27	18	55	40	3.06	2.22
fields.c	11150	6	38	31	6.33	5.17	8	19	21	2.38	2.62	3	18	6	6.00	2.00
grammar.lsp	3721	3	5	5	1.67	1.67	0	2	3	div 0	div 0	0	1	0	div 0	div 0
kennedy.xls	1029744	1323	34905	12829	26.38	9.70	1230	35962	13796	29.24	11.22	1541	4871	1671	3.16	1.08
lcet10.txt	426754	516	17151	6871	33.24	13.32	522	16447	6449	31.51	12.35	749	5815	2259	7.76	3.02
plrabn12.txt	481861	588	22657	8853	38.53	15.06	606	19295	9304	31.84	15.35	887	7841	2911	8.84	3.28
ptt5	513216	1248	7389	4617	5.92	3.70	696	5323	3705	7.65	5.32	1900	842	369	0.44	0.19
sum	38240	42	234	151	5.57	3.60	34	187	99	5.50	2.91	28	82	48	2.93	1.71
xargs.1	4227	6	25	9	4.17	1.50	2	6	2	3.00	1.00	2	4	2	2.00	1.00
LARGE CORPUS																
bible.txt	4047392	5055	337725	115481	66.81	22.84	5168	332777	116260	64.39	22.50	10285	158048	38038	15.37	3.70
E.coli	4638690	5534	382636	151405	69.14	27.36	6307	337196	151540	53.46	24.03	14808	140788	31189	9.51	2.11
world192.txt	2473400	3084	200643	67079	65.06	21.75	3089	187505	65213	60.70	21.11	5573	90738	25276	16.28	4.54
PURDUE CORPUS																
All_Up_1M.fasta	1001002	1238	61657	24597	49.80	19.87	1200	55389	23982	46.16	19.98	2350	14109	4167	6.00	1.77
All_Up_400k.fasta	399615	501	13959	6777	27.86	13.53	481	13294	6698	27.64	13.93	884	2758	1129	3.12	1.28
Helden_All.fasta	112507	119	1511	963	12.70	8.09	122	1363	933	11.17	7.65	165	382	191	2.32	1.16
Helden_CGN.fasta	32871	31	244	172	7.87	5.55	34	232	178	6.82	5.24	19	55	50	2.89	2.63
Spor_All_2x.fasta	444906	112	82	94	0.73	0.84	57	34	44	0.60	0.77	61	35	71	0.57	1.16
Spor_All.fasta	222453	246	3658	2107	14.87	8.57	250	3314	2140	13.26	8.56	413	775	401	1.88	0.97
Spor_EarlyI.fasta	31039	34	187	152	5.50	4.47	26	220	190	8.46	7.31	25	56	47	2.24	1.88
Spor_EarlyII.fasta	25008	20	145	151	7.25	7.55	15	166	121	11.07	8.07	33	60	39	1.82	1.18
Spor_Middle.fasta	54325	S 51	526	351	10.31	6.88	62	506	396	8.16	6.39	73	117	66	1.60	0.90

Figure 4. Comparison between update and reconstruction from scratch of the suffix array. Times are given in hundredth of seconds