Reconstructing a Suffix Array

F. Franek & W. F. Smyth

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario
Canada L8S 4K1

e-mail: franek@mcmaster.ca, smyth@mcmaster.ca

Abstract. For certain problems (for example, computing repetitions and repeats, data compression applications) it is not necessary that the suffixes of a string represented in a suffix tree or suffix array should occur in lexicographical order (lexorder). It thus becomes of interest to study possible alternate orderings of the suffixes in these data structures, that may be easier to construct or more efficient to use. In this paper we consider the "reconstruction" of a suffix array based on a given reordering of the alphabet, and we describe simple time- and space-efficient algorithms that accomplish it.

Keywords: suffix array, suffix tree, lexicographic order, alphabet, string

1 Introduction

We use a small example to introduce the main ideas. Consider the string

$$\begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \boldsymbol{x} = & a & b & a & a & b & \$ \end{array}$$

whose suffix tree $T_{\boldsymbol{x}}$ is shown in Figure 1 (the conventional sentinel \$ is a lexicographically least letter introduced to ensure that every suffix of $\boldsymbol{x}$ is represented as a leaf node of $T_{\boldsymbol{x}}$).

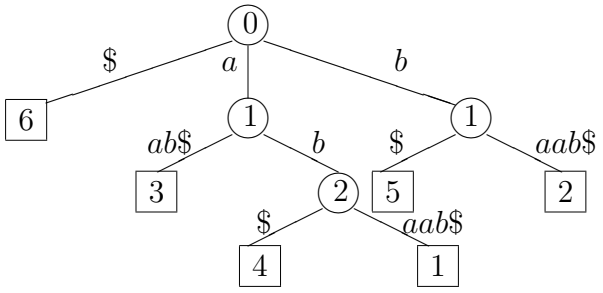


Figure 1: The suffix tree $T_{\boldsymbol{x}}$ of $\boldsymbol{x} = abaab$

Ignoring the sentinel suffix, a preorder traversal of $T_{\boldsymbol{x}}$ allows the suffix array of $\boldsymbol{x}$ to be read off in lexorder from the leaf nodes:

$$\begin{array}{rccccc} & 1 & 2 & 3 & 4 & 5 \\ \text{pos} = & 3 & 4 & 1 & 5 & 2 \end{array} \tag{1}$$

with the lengths (***lcp***s) of the corresponding longest common prefixes (***LCP***s) read off from the internal nodes:

$$\text{lcp} = 0 \quad 1 \quad 2 \quad 0 \quad 1. \tag{2}$$

Let us call the usual suffix array (for example, (1)) the ***lexicographical suffix array*** of $\boldsymbol{x}$ (LSA($\boldsymbol{x}$)), of course unique and well-defined for every string $\boldsymbol{x}$ on an ordered alphabet. More generally, we may define a ***valid suffix array*** of $\boldsymbol{x}$ (VSA($\boldsymbol{x}$)) to be any reordering of LSA($\boldsymbol{x}$) that can be obtained by reordering the subtrees of $T_{\boldsymbol{x}}$, then reading off the terminal nodes (except the sentinel suffix) in a preorder traversal. For our example string $\boldsymbol{x} = abaab$, there are actually 16 VSAs of $\boldsymbol{x}$:

$$\begin{array}{c} 34152,\ 34125,\ 31452,\ 31425 \\ 41352,\ 41325,\ 14352,\ 14325 \\ 52341,\ 25341,\ 52314,\ 25314 \\ 52413,\ 25413,\ 52143,\ 25143 \end{array}$$

Observe that of course for a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of length n, there are altogether $n!$ permutations of LSA($\boldsymbol{x}$); in our example 16 out of the $5! = 120$ permutations are actually VSAs. Note that if all the letters of $\boldsymbol{x}$ are distinct, then there will be $n!$ distinct VSAs of $\boldsymbol{x}$.

Finally we define a ***consistent suffix array*** of $\boldsymbol{x}$ (CSA($\boldsymbol{x}$)) to be a VSA that is determined by an ordering (reordering) of the alphabet. In our example, there are just two CSAs of $\boldsymbol{x}$:

$$34152 \text{ (for } \$ < a < b) \text{ and } 52413 \text{ (for } \$ < b < a).$$

In this paper we present algorithms to compute the CSA($\boldsymbol{x}$) determined by a specified ordering of the alphabet, given the LSA. As explained below, we think of this research as an initial step in gaining an understanding of how to compute a CSA or a VSA directly, without intermediate steps that depend on the LSA or the suffix tree.

Suffix arrays (LSAs) were introduced in 1990 [MM90, MM93] as a more space-efficient alternative to suffix trees; at the same time an $O(n \log n)$ algorithm was described for their construction. In 1997 a linear-time suffix *tree* construction algorithm was proposed [F97], effective in the normal case that the alphabet is ***indexed*** — that is, essentially, a finite integer alphabet. In 2003, based on [F97], three different groups of researchers independently discovered linear-time recursive algorithms to compute the LSA [KA03, KS03, KSPP03], also on an indexed alphabet. It turns out, however, that, largely as a consequence of their recursive nature, these algorithms are generally slower in practice [PST05] than two other classes of LSA construction algorithms whose worst-case behaviour is supralinear: ***direct comparison*** algorithms and ***prefix doubling*** algorithms. Direct comparison algorithms make use of a pointer copying method introduced in [BW94] to efficiently sort suffixes one letter at a time

[IT99, S00, MF04]; although their worst-case time requirement can therefore be as much as $\Theta(n^2 \log n)$, they generally have low space requirements and execute very fast in practice. On the other hand, prefix doubling algorithms make use of a technique introduced in [KMR72] to roughly double the length of the suffixes sorted at each step [MM93, LS99, BK03]; their worst-case time bound is thus only $O(n \log n)$ and they also tend to execute quickly in practice. Of the algorithms tested in [PST05], that of Manzini & Ferragina [MF04] appears to hold an advantage, both in the use of space and time, over that of Burkhardt & Kärkkäinen [BK03] in second place, but algorithms more recently described [SS05, M05] may be still more efficient.

The curious (to us, at least) fact is that to date the most efficient known way to compute any VSA is to first compute the LSA($\boldsymbol{x}$). In [FSXH03] we have described algorithms that essentially compute VSAs, but these algorithms are not as fast as the best LSA construction algorithms, even though LSA construction in general requires fewer conditions to be satisfied. It seems to us that VSA construction should be in some sense easier than LSA construction, but as things stand the opposite is true.

In this paper we will suppose that LSA($\boldsymbol{x}$) has been computed for $\boldsymbol{x} = \boldsymbol{x}[1..n]$ based on an ordering $(\mathcal{A}, <)$ of the alphabet $\mathcal{A}$. Then we show how to construct

$$\mathrm{CSA}(\boldsymbol{x}) = \mathrm{LSA}'(\boldsymbol{x})$$

determined by a reordering $(\mathcal{A}, <')$ of $\mathcal{A}$. In Section 2 we describe two $\Theta(n)$-time algorithms to handle a special case that arose in a recent paper [FS05]: ***reverse lexorder***, where for any letters $\lambda, \mu \in \mathcal{A}$,

$$\lambda < \mu \Longleftrightarrow \mu <' \lambda. \tag{3}$$

Section 3 presents an efficient algorithm for the general case: an arbitrary permutation of the order of the alphabet. Finally, Section 3 presents conclusions and outlines future work.

2 Reversing the Order of the Alphabet

As discussed in the Introduction, we assume that (3) holds, and we use LSA[1..n] to denote the suffix array corresponding to $(\mathcal{A}, <)$, LSA$'$[1..n] for the suffix array corresponding to $(\mathcal{A}, <')$. Recall that a ***border*** of a string $\boldsymbol{x}$ is any proper prefix of $\boldsymbol{x}$ that is also a suffix. We define the ***right border array*** $\beta = \beta[1..n]$ of $\boldsymbol{x}$ as follows: for every $i \in 1..n$, $\beta[i] = j \Longleftrightarrow j$ is the length of the longest border of $\boldsymbol{x}[i..n]$. β can be computed in $\Theta(n)$ time and constant space using a straightforward variant of the standard (left) border array algorithm [S03, ex. 1.3.10]. Observe that $\beta[i]$ is the lcp not only of $\boldsymbol{u} = \boldsymbol{x}[n\beta[i]+1..n]$ and $\boldsymbol{v} = \boldsymbol{x}[i..n]$, but also of every suffix $\boldsymbol{w}$ of $\boldsymbol{x}$ that lies between $\boldsymbol{u}$ and $\boldsymbol{v}$ in lexorder.

For technical reasons to simplify the presentation of the following lemmas and algorithms, we modify slightly the array β: $\beta[i] \neq 0$ is not the length of the longest border of $\boldsymbol{x}[i..n]$, but the index of the suffix of $\boldsymbol{x}$ that is the longest border, i.e. $\beta[i] = j \neq 0$ if and only if $\boldsymbol{x}[j..n]$ is the longest border of $\boldsymbol{x}[i..n]$ (see Figure 2).

The algorithms for reverse lexorder are then a consequence of the following lemmas:

```
β[1] ← 0;
for i ← 1 to n−1 do
  if β[n−i+1] = 0 then
    c ← 0
  else
    c ← n+1−β[n−i+1]
  while c > 0 and x[n−i] ≠ x[n−c]
    if β[n−c+1] = 0 then
      c ← 0
    else
      c ← n+1−β[n−c+1]
  if x[n−i] = x[n−c] then
    β[n−i] ← n−c
  else
    β[n−i] ← 0
```

Figure 2: Computing $\beta[1..n]$ for input string $\boldsymbol{x}$[1..n]

Lemma 1 *Let* $j = \mathrm{LSA}[i]$ *for some* $i \in 1..n$.

(a) *If* $\beta[j] > 0$, *then* $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$;

(b) *otherwise, if* $\beta[j] = 0$, *then*

$$\boldsymbol{x}[j..n] <' \min_{1 \le h < i} \boldsymbol{x}\big[\mathrm{LSA}[h]..n\big]. \tag{4}$$

Proof If $\beta[j] > 0$, then $\boldsymbol{x}\big[\beta[j]..n\big]$ is a proper prefix of $\boldsymbol{x}[j..n]$, so that $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$. If $\beta[j] = 0$, then for every $h \in 1..i1$, there exists a least nonnegative integer $q_h \le \min\{nj+1, n\mathrm{LSA}[h]+1\}$ such that $\boldsymbol{x}\big[\mathrm{LSA}[h]+q_h\big] \ne \boldsymbol{x}[j+q_h]$. Thus by the definition of LSA, $\boldsymbol{x}\big[\mathrm{LSA}[h]+q_h\big] < \boldsymbol{x}[j+q_h]$, and so, by the definition of $<'$, $\boldsymbol{x}[j+q_h] <' \boldsymbol{x}\big[\mathrm{LSA}[h]+q_h\big]$. Hence (4) holds. □

Observe that every border of every suffix is represented by an entry in β and so will be covered by Lemma 1. Observe further that the quantities q_h introduced in the proof for $\beta[j] = 0$ are actually lcp values for each pair of suffixes $\boldsymbol{x}[j..n]$ and $\boldsymbol{x}\big[\mathrm{LSA}[h]..n\big]$.

Lemma 2 *Let* $j_1 = \mathrm{LSA}[i_1]$, $j_2 = \mathrm{LSA}[i_2]$, $1 \le i_1 < i_2 \le n$. *If* $\beta[j_1] = \beta[j_2] > 0$, *then*

$$\boldsymbol{x}[j_2..n] <' \boldsymbol{x}[j_1..n].$$

Proof Since $i_1 < i_2$, $\boldsymbol{x}[j_1..n] < \boldsymbol{x}[j_2..n]$; since neither of these strings can be a prefix of the other, the result follows. □

Figure 3 shows the simplest algorithm that computes LSA′. The algorithm illustrates the fundamental idea of the process in a clear and simple way. We suppose that the array β was computed in preprocessing, while the array NEXT[1..n] emulates a singly-linked list equivalent to LSA′ that is constructed as the input LSA is

```
start ← LSA[1];
for i ← 2 to n do
  j ← LSA[i]
  if β[j] = 0 then
    — by Lemma 1 (b) j goes to start of list
    NEXT[j] ← start; start ← j
  else
    — by Lemmas 1 (a) & 2, insert j next to β[j]
    j′ ← β[j]; temp ← NEXT[j′]
    NEXT[j′] ← j; NEXT[j] ← temp
```

Figure 3: Algorithm 1 — Computing LSA′ for Reversed Alphabet

scanned from left to right (in increasing lexorder): we will consistently use the word ***transform*** to refer to the computation of NEXT from LSA (and *vice versa*). We omit the straightforward **for** loop that transforms NEXT into LSA′.

```
 — transform LSA into NEXT
start ← LSA[1]
for i ← 1 to n−1 do
  NEXT[LSA[i]] ← LSA[i+1]
NEXT[LSA[n]] ← 0
compute β using memory storage of LSA
 — reorder NEXT
prev ← start; cur ← NEXT[prev]
while cur ≠ 0 do
  if β[cur] = 0 then  — cur goes to front
    NEXT[prev] ← NEXT[cur]; NEXT[cur] ← start
    start ← cur
  else  — cur goes next to β[cur]
    if NEXT[β[cur]] = cur then
      prev ← cur
    else
      NEXT[prev] ← NEXT[cur]; i ← NEXT[β[cur]];
      NEXT[β[cur]] ← cur; NEXT[cur] ← i
 — transform NEXT to LSA′ using memory storage of β
i ← 1; j ← start
for i ← 1 to n do
  LSA[i] ← j; j ← NEXT[j]
```

Figure 4: Algorithm 2 — Computing LSA′ for Reversed Alphabet

Algorithm 1 has the disadvantage of using $2|\boldsymbol{x}|$ words of working memory (the arrays β and NEXT) for the input string $\boldsymbol{x}$. Algorithm 2 (see Figure 4) is a bit more elaborate; however, it is based on the same principles as Algorithm 1 and uses only $|\boldsymbol{x}|$ words of working memory (for NEXT).
Thus

Theorem 1 *Given* LSA($\boldsymbol{x}$) *for a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*, Algorithm 2 computes* LSA$'(\boldsymbol{x})$ *for a reversed alphabet in* $\Theta(n)$ *time using* n *words of working memory.*

Proof By induction. Clearly for $i = 1$ the entries in NEXT are in $<'$ order. Suppose that for arbitrary $i \in 1..n1$, the entries are in $<'$ order. By Lemmas 1 and 2, the entries must still be in $<'$ order after LSA$[i+1]$ has been processed. □

We note that essentially the same algorithm applies to a morphism $\sigma : \mathcal{A} \to \mathcal{B}$ from one ordered alphabet to another provided that for every distinct $\lambda, \mu \in \mathcal{A}$, $\lambda < \mu \iff \sigma(\mu) <' \sigma(\lambda)$.

3 Permuting the Order of the Alphabet

In this section we describe an algorithm to compute LSA$'(\boldsymbol{x})$ in the case of an arbitrary reordering $(\mathcal{A}, <')$ of the alphabet $\mathcal{A}$. Alternatively, we may think of this reordering as a permutation $\pi : \mathcal{A} \to \mathcal{A}$ where for every distinct $\lambda, \mu \in \mathcal{A}$, $\lambda < \mu \iff \pi(\lambda) <' \pi(\mu)$.

Essentially, our algorithm uses LSA($\boldsymbol{x}$) (in fact, as we shall see, any VSA($\boldsymbol{x}$) will do) to simulate a reordering of the subtrees of the suffix tree $T_{\boldsymbol{x}}$ that is determined by the reordering of the alphabet. In the simple example of Figure 1, the only possible reordering (since $|\mathcal{A}| = 2$, necessarily a reversal) would result from interchanging two paths in the subtree represented by a and b as well as in the subtree represented by $aab\$$ and ab, yielding LSA$'(\boldsymbol{x}) = 52413$.

It is instructive to consider the relationship between reversal and arbitrary reordering. In Lemma 1, if we suppose that $\beta[j] > 0$, it is true also in the general case that $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$; however, Lemmas 1 (b) and 2 no longer hold, since it is no longer possible to infer the order of $\boldsymbol{x}[j_1..n]$ and $\boldsymbol{x}[j_2..n]$ from the order in which they occur in LSA($\boldsymbol{x}$). In other words, the set of suffixes that have the same LCP $\boldsymbol{x}\big[\beta[j]..n\big]$ cannot simply be placed to the right of $\boldsymbol{x}\big[\beta[j]..n\big]$ — they must now be sorted in $<'$ order based on positions $\beta[j]+1, \beta[j]+2, \ldots$ in each suffix.

Similarly, in the case that $\beta[j] = 0$, (4) no longer holds: we must relocate suffixes by sorting in $<'$ order the ones that have the same LCP (occur in the same subtree of $T_{\boldsymbol{x}}$).

These comments imply that the array β is no longer useful in the general case, whereas the lcp array (for example, (2)) becomes critical. Fortunately, like β, the lcp array lcp$[1..n]$ can be computed in linear time, either from the LSA [KLAAP01] or as a byproduct of LSA construction: thus we assume throughout this section that it is available. In fact, as noted above, since in the general case the LSA ordering provides no information about the LSA$'$ ordering, the algorithm described in this section will work just as well using any VSA($\boldsymbol{x}$) together with its corresponding (permuted) lcp array.

Our algorithm reorders the suffixes of $\boldsymbol{x}$ beginning with those that share the greatest lcp values, thus equivalent to a traversal of the suffix tree $T_{\boldsymbol{x}}$ upwards from the deepest lcp nodes. We first outline the control structure that our algorithm uses to accomplish this traversal, then go on to describe the details of its implementation.

The input LSA($\boldsymbol{x}$) (VSA($\boldsymbol{x}$)) and its corresponding input lcp array LCP1 are being traversed from left to right in order to identify ***families***. In simple terms, a family is a set of nodes in the lis NEXTthat corresponds to a set of links to nodes that are

immediate children of an internal node of the corresponding suffix tree. These links can be permuted provided that the links in all subtrees have been already sorted. If the internal node that is the root of the subtree corresponds to lcp ℓ, we call the family an **ℓ-family**. A stack STACK for tracking families is maintained by the algorithm; if a value ℓ is on top of the stack, then an LCP[NEXT[ℓ]]-family starts at position NEXT[ℓ] (for technical reason we do not store the beginning of the family on the stack, but rather the previous node).

```
 — input: x - string
 — input: LSA- its suffix array
 — input: LCP1- lcp array for LSA
 — input: permutation p of the alphabet
NEXT[ ]  — auxiliary array
STACK  — stack for keeping track of families
Transform LSA to NEXT
Transform LCP1 to LCP using memory of LSA for LCP
use memory of LCP1 as memory for TAIL and initialize it
Initialize STACK and variables
while multipop()
  Identify and Extract a family (using STACK)
  Sort the family (using p)
  Flatten the family
  Verticalize the family
Sort the final 0-family
Flatten the final 0-family
Transform NEXT to LSA
Transform LCP to LCP1
 — output: LSA sorted according to p
 — output: LCP1 lcp of LSA
```

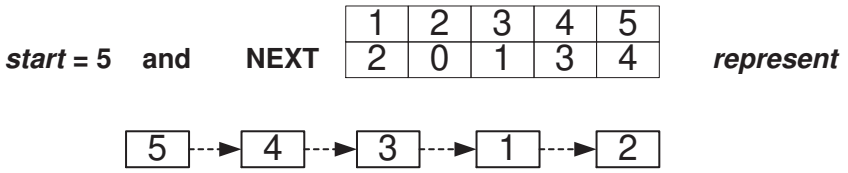
Figure 5: Outline of Algorithm 3 — General Reordering

The families are identified simply during the scan: as long as the values of LCP increase, they are pushed on the stack as they represent beginnings of families. A decreasing value indicates the end of the innermost family (i.e. the one on the top of the stack). After the family is sorted, it is "verticalized", so it is now represented as a single node in the family it is nested in and the scan can continue. One would expect to pop the stack once the innermost family is processed. However, the situation is a bit more complex, and thus ***multipop***() is employed to decide whether or not the stack should be popped. The control stucture of the algorithm is shown in Figure 5. The individual steps are described in detail below, making use of the following standard routines: $Push(s)$ pushes s on top of STACK, $Pop()$ pops STACK, $Top()$ obtains the value on the top of the stack STACK without popping it, $Top_1()$ obtains the value next to the top of STACK without popping it.

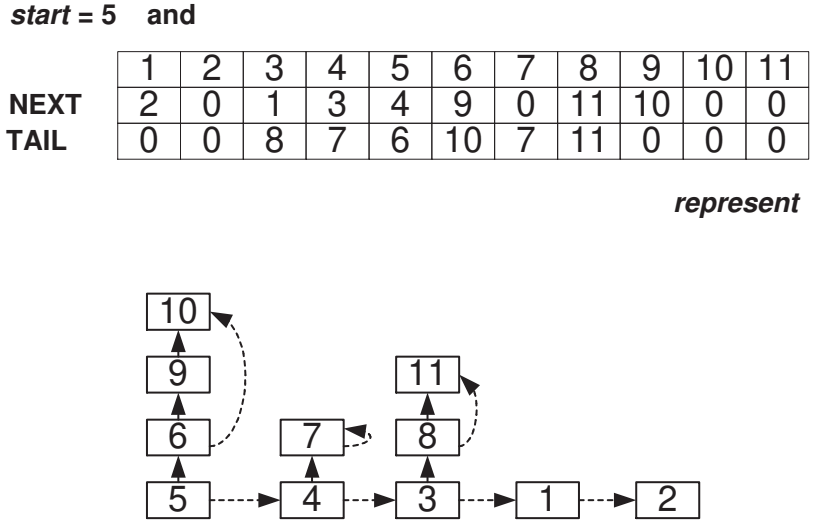
The data structures and variables

As shown in Figure 5, three arrays are used in addition to $\boldsymbol{x}$, two of them input, only one auxiliary. NEXT[1..n] emulates a singly-linked list of nodes, where each node

stores an integer value k representing the suffix $\boldsymbol{x}[k..n]$. A variable *start* marks the beginning of NEXT[ ]. For instance,



The array TAIL[n] represents the "verticalized" part of the list of nodes. For instance,



The end of each "vertical" tail is reachable in two steps: TAIL[TAIL[k]] is the very last member of the "vertical" tail starting at the node k. Other auxiliary variables used are: *cur* for a "pointer" to the current node in NEXT[ ], *prev* for a "pointer" to the previous node (if $prev = 0$, it means that $cur = start$). LE (*left end*) represents the node to which the head of a family is attached ($LE = 0$ means that the head of the family is *start*), RE (*right end*) represents the node to which the last member of a family will point to ($RE = 0$ means that the last member of a family is the last member of the NEXT list). Finally a variable *type* describes the type of family we are processing, i.e. the lcp of all memebers of the family.

Transform input LSA **to** NEXT

Traverse LSA and fill in the entries in NEXT:

$start \leftarrow$ LSA[1];
for $i \leftarrow 1$ **to** $n-1$ **do**
  NEXT[LSA[i]] $\leftarrow$ LSA[$i+1$]
NEXT[n] $\leftarrow 0$

Transform input LCP1 **to** LCP

Normally, LCP[i] represents the lcp of two neighbouring suffixes, $\boldsymbol{x}$[LSA[$i-1..n$]] and $\boldsymbol{x}$[LSA[$i..n$]]. But since during the sorting the mutual positions of suffixes can change, we modify the usual meaning to: LCP[i] represents the lcp of $\boldsymbol{x}$[LSA[$i..n$]] and its right neighbour. Thus, we traverse LCP1 and "shift" the values one position to the left. Since LSA is no longer needed, we use its memory for LCP:

```
LCP[start] ← LCP1[1];
for i ← 1 to n−1 do
  LCP[LSA[i]] ← LCP1[i+1]
LCP[n] ← 0
```

Initialize TAIL

Since LCP1 is no longer needed, we use its memory for TAIL. Since at the beginning we have no "vertical" tails, all entries must be initialized to 0:

```
for i ← 1 to n do
  TAIL[i] ← 0
```

Initialize STACK **and variables**

Start the traversal of NEXT and LCP. Keep traversing as long as LCP has value 0. Push on STACK *prev* of the first non-zero node.

```
prev ← 0; cur ← start
while LCP[i] = 0
  prev ← cur; cur ← NEXT[cur]
Push(prev)
type ← LCP[cur]
```

Identify and Extract a family

Note that we are now inside a loop (see Figure 5), and thus the use of the term **continue** means to transfer the flow of control to the top of the loop.

```
if LCP[cur] = type then
  prev ← cur; cur ← NEXT[cur]; continue
if LCP[cur] > type then  — a new family starts
  Push(prev)
  prev ← cur; cur ← NEXT[cur]; continue
if LCP[cur] < type then  — a family ends
  LE ← Top(); RE ← NEXT[cur]; NEXT[cur] ← 0
```

Thus we have just identified an innermost family of type LCP[*cur*] starting at NEXT[*LE*] and ending at *cur*. Note that we "severed" the link between *cur* and *RE* (we "extracted" the family from the list NEXT).

Sort the family

Note that sorting the family according to the letter at position *type* is the same as sorting links of an internal node of a suffix tree. We will discuss the actual sorting separately. We are assuming that *from* refers to the head of the family, while *to* to its last member. Prior to sorting the family, we must remember the LCP[*to*] value, thus *last* ← LCP[*to*]. After the sorting of the family, we must modify the LCP accordingly:

```
for i ← from to to
  if LCP[i] < type then
    LCP[i] ← type
LCP[to] → last
```

Flatten the family

As indicated, some nodes in the NEXT list might have "vertical" tails. At this stage we "flatten" the family so there are no "vertical" tails any more. The process is simple: if NEXT[a] = b, then we make NEXT[a] to be the first element in the "vertical" tail, while NEXT[c] $\leftarrow b$, where c is the last element in the "vertical" tail. Thus:

for $i \leftarrow from$ **to** to
  if TAIL[i] $\neq 0$ **then**
    $b \leftarrow$ NEXT[i]; NEXT[i] $\leftarrow$ TAIL[i]
    NEXT[TAIL[TAIL[i]]] $\leftarrow b$
    TAIL[TAIL[i]] $\leftarrow 0$; TAIL[i] $\leftarrow 0$

Verticalize the family

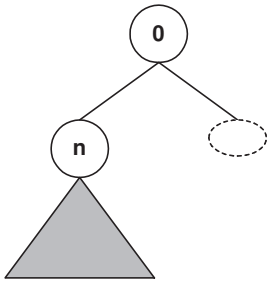
To prevent resorting or retraversing the family which just has been flattened during the subsequent sort (of the family this family is nested in), we leave only the head of the family in the NEXT list, and make the rest of the family into a "vertical" tail of the head. Thus, in all subsequent sorts only the head will be used and thus further traversal of the family is prevented.

TAIL[$from$] = NEXT[$from$]
NEXT[$from$] $\leftarrow 0$
TAIL[TAIL[$from$]] $\leftarrow to$

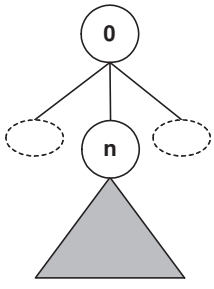
***multipop*()**

As a technicality, in its first invocation **multipop()** returns **true** . Thus, we can assume, that we just finished processing a family of type *type*. We have to decide if we continue with the scan, pop the stack, or process another family. The role of ***multipop***() is to make all these decisions. It returns **true** if the scan is to continue, or **false** if the scan is to terminate.

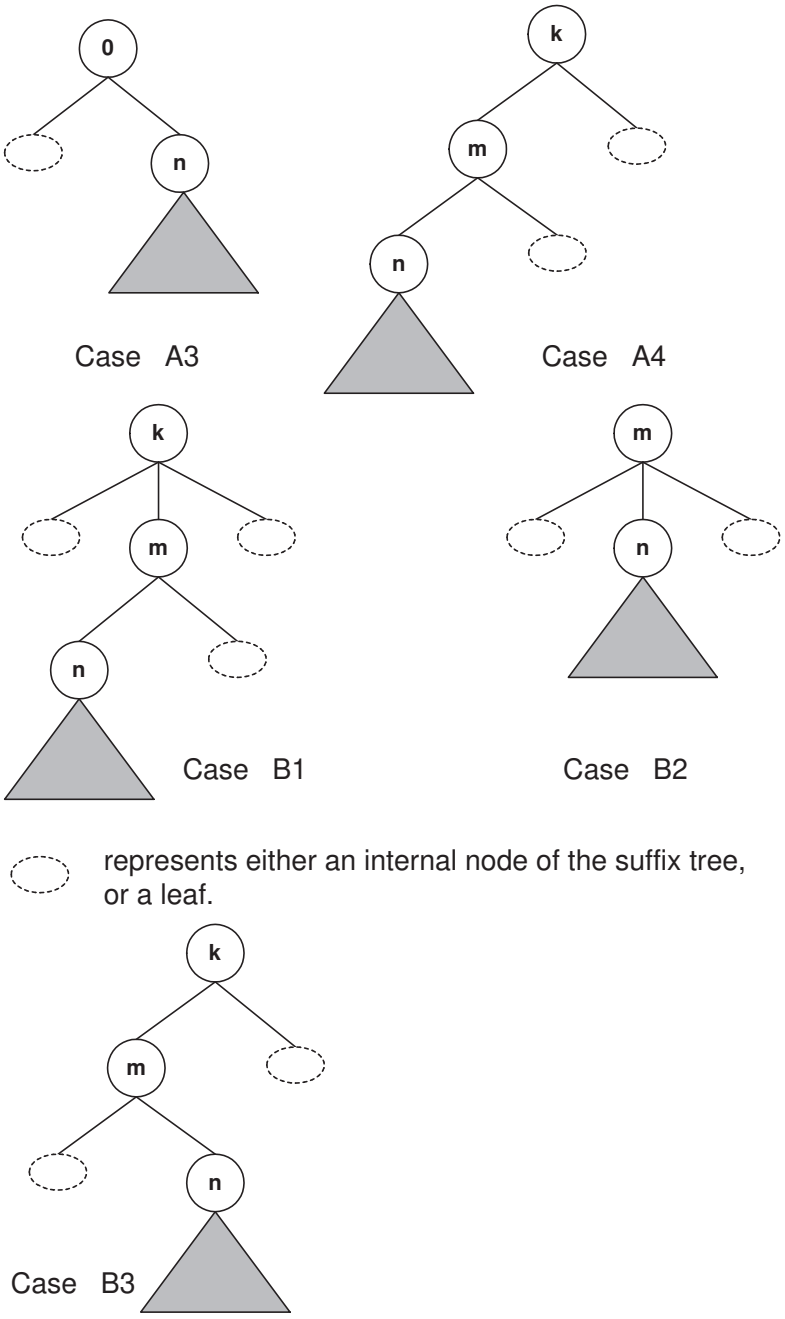
What situations can happen is best visualized on the suffix tree — the grey triangle represents the family of links that was just sorted. There are 7 possible cases that we denote A1, ..., A4, and B1, ..., B3. Cases A1, ..., A4 concern situations when only one item is on the stack (representing the family we just sorted), while cases B1, ..., B3 concern situations when more than one item are on the stack. The schematic depiction of the cases follows:



Case A1



Case A2



The variable $famend$ represents the "pointer" to the very last element in the family just processed.

Cases A1, ..., A3
These are treated alike and recognized alike. The recognition is based on the fact that the stack has only one item and LCP[$famend$] = 0. The action is to pop the stack, forward the scan and then the scan is continued:

$Pop()$; $prev \leftarrow cur$; $cur \leftarrow$ NEXT[cur]
if cur=0 **then return false**
$type \leftarrow$ LCP[cur]
while $type = 0$ **do**
  $prev \leftarrow cur$; $cur \leftarrow$ NEXT[cur]; $type \leftarrow$ LCP[cur]
  if type=0 **then**
    $prev \leftarrow cur$; **return false**
$Push(prev)$
return true

Case A4
The recognition is based on the fact that the stack has only one item and LCP[$famend$] > 0. The action is not to pop the stack (as the n-family just processed starts at the same position as the m-family to be processed), the scan is forwarder and then the scan is continued, but the type is decreased accordingly (to m):

$type \leftarrow$ LCP[$famend$]
$prev \leftarrow cur$
$cur \leftarrow$ NEXT[cur]
return true

For cases B1, ..., B3 we have to determine $type1$, the type of the family that is on the top of the stack:

if $Top_1() = 0$ **then**
  $type1 \leftarrow$ LCP[$start$]
else
  $type1 \leftarrow$ LCP[NEXT[$Top_1()$]]

Case B1
The recognition is based on the fact that the stack has more than one item and LCP[$famend$] $> type1$. The action is not to pop the stack (as the n-family just processed starts at the same position as the m-family to be processed), the scan is forwarded and then the scan is continued, but the type is decreased accordingly (to m):

$type \leftarrow$ LCP[$famend$]
$prev \leftarrow cur$
$cur \leftarrow$ NEXT[cur]
return true

Case B2
The recognition is based on the fact that the stack has more than one item and LCP[$famend$] $= type1$. The action is to pop the stack, decrease the type, forward the scan and then the scan is continued:

$Pop()$
$type \leftarrow type1$
$prev \leftarrow cur$
$cur \leftarrow$ NEXT[cur]
if $cur = 0$ **then return false**
return true

Case B3
The recognition is based on the fact that the stack has more than one item and LCP[$famend$] < $type1$. The action is to pop the stack, decrease the type, without moving forward the scan and then the scan is continued:

$Pop()$
$type \leftarrow type1$
return true

This concludes the description of the algorithm. It is rather straighforward to check that the algorithm (without the actual sorting of the families) requires $O(n)$ steps. The additional memory requirements are n words for the array NEXT[ ] and $\leq n$ words of memory for STACK. Of course, some additional memory will be required for the actual sorting of the families: if the number of distinct characters in the input string is $\leq n/2$, then we need $\leq 3n/2$ words of memory for STACK and for sorting (n for STACK and $\leq n/2$ for sorting). If the number of distinct characters in the input string is $> n/2$, then we need $\leq 3n/2$ words of memory for STACK and sorting ($< n/2$ for STACK, and $\leq n$ for sorting). Thus, **the algorithm presented requires in total $\leq 2.5n$ words of working memory for the process and the sorting**.

C code for Algorithms 1–3 and powerpoint illustration of Algorithms 2–3 are available at [F05].

From the presentation of the algorithm it is clear that sorting the suffix array is as complex as sorting links in the corresponding suffix tree. Thus, the following discusion applies to both suffix trees and suffix arrays. When we are to sort a family of size k (or k links of an internal node in the suffix tree), no matter what permutation is given, it can be sorted in $O(n)$ time using a bucket sort. However, this may lead to non-linear sorting time for the whole array (or the whole tree). If the alphabet is fixed, of course the sorting will be linear. But also for some "mild" permutations the sorting will be linear as well. This leads us to investigate an interesting computational property of permutations that we call the suborder complexity of the permutation:

The ***suborder complexity*** β of a permutation p of n, denoted $\beta(p)$, is defined to be the minimal β such that for any $2 \leq k \leq n$, it takes at most βk steps to order any subset of n of size k. Note that $\beta(p) \leq \log n$ as any subset of n of size k can be sorted in $\leq k \log k \leq k \log n$ steps.

It follows that

Theorem 2 *For any permutation with suborder complexity β, the suffix array of a string can be re-ordered by Algorithm 3 in $O(\beta n)$ time, where n is the length of the input string.*

Conclusions and Further Research

An interesting question that arises is what kind of permutations have small suborder complexity. Here are some examples:

- The inversion has suborder complexity 1.

- Any rotation has suborder complexity 1.
- Any permutation with β transpositions has suborder complexity β.
- Let p be a "mild" permutation, i.e. $|p(i) - i| \leq \beta$. Then p has suborder complexity 2β.
- Let p_1 on n_1 have suborder complexity β_1 and let p_2 on n_2 have suborder complexity β_2, then $p_1 \bigoplus p_2$ has suborder complexity $\max(\beta_1, \beta_2)$ (where $p = p_1 \bigoplus p_2$ is defined on n_1+n_2 by $p(i) = p_1(i)$ for $1 \leq i \leq n_1$, and $p(i) = n_1+p_2(i-n_1)$ for $n_1 < i \leq n_1+n_2$).

So the class of permutations with small suborder complexity seems quite interesting and rich enough to warrant further investigation.

Acknowledgements

The research was supported in part by the authors' research grants from the Natural Sciences and Engineering Research Council of Canada.

References

[BK03] S. Burkhardt & J. Kärkkäinen, **Fast lightweight suffix array construction and checking**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 55-69.

[BW94] M. Burrows & D.J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Research Report 124, Digital Equipment Corporation (1994) 18 pp.

[F97] M. Farach, **Optimal suffix tree construction with large alphabets**, in *Proc. 38th Annual Symp. Foundations of Computer Science*, IEEE (1997) 137–143.

[F05] F. Franek, *C code + illustration*:

`http://www.cas.mcmaster.ca/~franek/web-publications.html`

[FS05] F. Franek & W. F. Smyth, **Sorting the suffixes of a two-pattern string**, *Internat. J. Foundations of Computer Sci.* (2005) to appear.

[FSXH03] F. Franek, W. F. Smyth, X. Xiao & J. Holub, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[IT99] H. Itoh & H. Tanaka, **An efficient method for in memory construction of suffix arrays**, *Proc. String Processing & Information Retrieval Symp.*, IEEE (1999) 81–88.

[KLAAP01] T. Kasai, G. Lee, H. Arimura, S. Arikawa & K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181-192.

[KA03] P. Ko & S. Aluru, **Space Efficient Linear Time Construction of Suffix Arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 200–210.

[KMR72] R. M. Karp, R. E. Miller & A. L. Rosenberg, **Rapid identification of repeated patterns in strings, trees and arrays**, *Proc. 4th Annual ACM Symp. on Theory of Computing* (1972) 125–136.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, & K. Park, **Linear-time Construction of Suffix Arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 186–199.

[KS03] J. Kärkkäinen & P. Sanders, **Simple Linear Work Suffix Array Construction**, *Proc. 30th International Colloqium on Automata, Languages and Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[LS99] N. Jesper Larsson & K. Sadakane, *Faster Suffix Sorting*, Technical Report LU-CS-TR:00–214, Lund University (1999) 20 pp.

[MM90] U. Manber & G. Myers, **Suffix Arrays: A new method for on-line string searches**, *Proc. First ACM-SIAM Symp. on Discrete Algs.* (1990) 319–327.

[MM93] U. Manber & G. Myers, **Suffix Arrays: A new method for on-line string searches**, *SIAM J. Computing 22* (1993) 935–948.

[M05] M. Maniscalco, *MSufSort*:

`http://www.michael-maniscalco.com/`

[MF04] G. Manzini & P. Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[PST05] S. J. Puglisi, W. F. Smyth & A. Turpin, **The performance of linear time suffix sorting algorithms**, *Proc. Data Compression Conf. '05* (2005) to appear.

[S00] J. Seward, **On the performance of BWT sorting algorithms**, *Proc. Data Compression Conf. '00* (2000) 173–182.

[SS05] K. Schürmann & J. Stoye, **An incomplex algorithm for fast suffix array construction**, *Proc. 7th Workshop Algorithm Engineering & Experiments* (2005) to appear.

[S03] B. Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) pp. 423.