9 Enhanced Suffix Arrays

This exposition by Clemens Gröpl is based on the following sources, which are all recommended reading:

1. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2 (2004) 53-86.
2. Kasai, Lee, Arimura, Arikawa, Park: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, CPM 2001

9.1 Introduction

The term *enhanced suffix array* stands for data structures consisting of a suffix array and additional tables. We will see that every algorithm that is based on a *suffix tree* as its data structure can systematically be replaced with an algorithm that uses an enhanced suffix array and solves the same problem in the same time complexity. Very often the new algorithms are not only more space efficient and faster, but also easier to implement.

Suffix trees have many uses. These applications can be classified into three kinds of tree traversals:

1. A bottom-up traversal of the complete suffix tree.
2. A top-down traversal of a subtree of the suffix tree.
3. A traversal of the suffix tree using suffix links.

An example for bottom-up traversal is the MGA algorithm (MGA = multiple genome aligment).

An example for top-down traversal is exact pattern matching. We have seen that the trivial search can be improved from $O(m \log n)$ to $O(m + \log n)$ if the suffix array is "enhanced" by an *lcp* table.

9.2 Repeats vs. repeated pairs

Let us recall some definitions and fix terminology. Let S be the underlying sequence for the suffix array and $n := |S|$.

- A pair of substrings $R = ((i_1, j_1), (i_2, j_2))$ is a ***repeated pair*** iff $(i_1, j_1) \neq (i_2, j_2)$ and $S[i_1..j_1] = S[i_2..j_2]$. The *length* of R is $j_1 - i_1 + 1$.
- R is *left maximal* iff $S[i_1 - 1] \neq S[i_2 - 1]$ (i. e., the "left characters" disagree).
- R is *right maximal* iff $S[j_1 + 1] \neq S[j_2 + 1]$ (i. e., the "right characters" disagree).
- R is *maximal* iff it is left maximal and right maximal.
- A substring ω of S is a ***repeat*** iff there is a repeated pair R whose consensus is $\omega = S[i_1..j_1]$.
- Then ω is *maximal* iff R is.
- A *supermaximal repeat* is a maximal repeat that never occurs as a substring in any other maximal repeat.

9.3 The basic tables

- ***suftab***: The suffix table. An array of integers in the range 0 to n, specifying the lexicographic ordering of the $n + 1$ suffixes of the string $S\$$. Requires $4n$ bytes.
- ***sufinv***: The inverse of the suffix table. An array of integers in the range 0 to n such that $sufinv[suftab[i]] = i$. Can be computed in linear time from *suftab*. Requires $4n$ bytes.

- ***lcptab***: Table for the length of the longest common prefix for consecutive entries of *suftab*: $lcptab[0] := 0$ and $lcptab[i] := lcp(S_{suftab[i]}, S_{suftab[i-1]})$ for $1 \le i \le n$. Aka. the *height* array. Can be computed in linear time from *suftab* and *sufinv* using the algorithm of Kasai et al.. Requires $4n$ bytes in the worst case, but usually can be "compressed" to $(1+\varepsilon)n$ bytes.
- ***bwttab***: The *Burrows and Wheeler transformation* of S. Known from data compression (e. g. `bzip2`). Contains the character preceding the suffix stored in *suftab*: $bwttab[i] := S_{suftab[i]-1}$ if $suftab[i] \neq 0$, undefined otherwise. Can be computed in linear time from *suftab*. Requires n bytes.

9.4 Lcp-intervals

Definition. Let $0 \le i < j \le n$. Then $[i..j]$ is an *lcp-interval of lcp-value* ℓ iff:

1. $lcptab[i] < \ell$
2. $lcptab[k] \ge \ell$ for all k with $i < k \le j$
3. $lcptab[k] = \ell$ for at least one k with $i < k \le j$
   (Such a k is called an *ℓ-index*.)
4. $lcptab[j+1] < \ell$

Such an $[i..j]$ will also be called an *ℓ-interval* or even just "an ℓ-$[i..j]$".

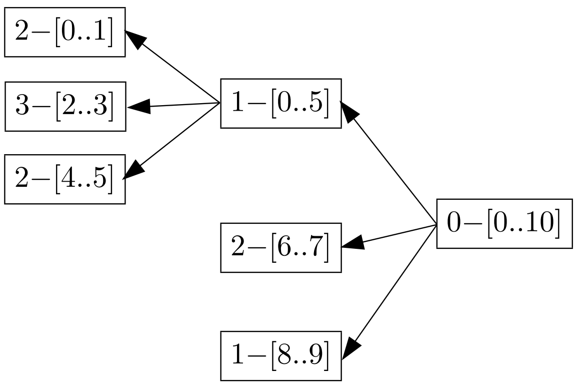
The idea behind ℓ-intervals is that they correspond to internal nodes of the suffix tree.

The set of all ℓ-indices of an ℓ-interval $[i..j]$ is denoted by $\ell Indices(i,j)$.

$[i..j]$ is the *ω-interval*, where ω is the longest common prefix of $S_{suftab[i]}, \dots, S_{suftab[j]}$.

The example below shows the lcp-interval tree implied by the *suftab* and *lcptab* tables.

i	*suftab*	*lcptab*	*bwttab*	$S_{suftab[i]}$
0	2		`c`	`aaacatat$`
1	3	2	`a`	`aacatat$`
2	0	1		`acaaacatat$`
3	4	3	`a`	`acatat$`
4	6	1	`c`	`atat$`
5	8	2	`t`	`at$`
6	1	0	`a`	`caaacatat$`
7	5	2	`a`	`catat$`
8	7	0	`a`	`tat$`
9	9	1	`a`	`t$`
10	10	0	`t`	`$`



9.5 Supermaximal repeats

We are now ready to characterize (and in fact, compute) supermaximal repeats using the basic tables *suftab*, *lcptab*, and *bwttab*.

Definition. An ℓ-interval is a *local maximum* iff ℓIndices$(i,j) = [i..j]$, that is, $lcptab[k] = \ell$ for all $i < k \le j$.

Lemma. A string ω is a supermaximal repeat iff there is an ℓ-interval $[i..j]$ such that

1. $[i..j]$ is a local maximum and $[i..j]$ is the ω-interval, and
2. the "left" characters $bwttab[i], \dots, bwttab[j]$ are pairwise distinct.

Proof. It is easy to check that these conditions are necessary and sufficient for ω being a supermaximal repeat. (Exercise)

Clearly we can find all local maxima in one scan trough *lcptab*. Thus we already have an algorithm to enumerate all supermaximal repeats! Note that by the preceeding lemma, there are at most n of them.

9.6 The (virtual) lcp-interval tree

We have already seen in the example that the lcp-intervals are nested.

Definition. We say that an m-interval $[l, r]$ is *embedded* in an ℓ-interval $[i, j]$ iff $i \leq l < r \leq j$ and $m > \ell$. We also say that $[i..j]$ *encloses* $[l..r]$ in this case.

If $[i..j]$ encloses $[l, r]$, and there is no (third) interval embedded in $[i..j]$ that also encloses $[l..r]$, then $[l..r]$ is called a *child interval* of $[i..j]$. Conversely, $[i..j]$ is called the *parent interval* of $[l..r]$ in this case.

The nodes of the lcp-interval tree are in one-to-one correspondence to the internal nodes of the suffix tree. But the lcp-interval tree is not really built by the algorithm. It is only a useful concept, similar to a depth-first-search tree.

9.7 Bottom-up traversals

In order to perform a bottom-up traversal, we keep the nested lcp-intervals on a stack. The operations *push*, *pop*, and *top* are defined as usually. The elements on the stack are the lcp-intervals represented by tuples (lcp, lb, rb): lcp is the lcp-value of the interval, lb is its left boundary, and rb is its right boundary. Furthermore, $\perp$ denotes an undefined value.

The following algorithm reports all lcp-intervals. (Abuelhoda et al. [AKO04], adapted from Kasai et al. [KLAAP01], the last line is a fix [cg] to report the root.)

Algorithm 1. (Bottom-up traversal)

```
$push((0, 0, \perp))$
for $i := 1$ to $n$ do
    $lb := i - 1$
    while $lcptab[i] < top.lcp$
        $top.rb := i - 1$
        $interval := pop$
        $report(interval)$
        $lb := interval.lb$
    if $lcptab[i] > top.lcp$ then
        $push((lcptab[i], lb, \perp))$
$top.rb = n$; $interval := pop$; $report(interval)$
```

Example. We sketch the run for $S = acaaacatat\$$:

0. *Init stack.* push(0,0,⊥)
1. while: No. if: Yes. ⇒ push(2,0,⊥)
2. while: Yes. ⇒ report(2,0,1)
   while: No. if: Yes. ⇒ push(1,0,⊥)
3. while: No. if: Yes. ⇒ push(3,2,⊥)
4. while: Yes. ⇒ report(3,2,3)
   while: No. if: No.
5. while: No. if: Yes. ⇒ push(2,4,⊥)
6. while: Yes. ⇒ report(2,4,5)
   while: Yes. ⇒ report(1,0,5)
   while: No. if: No.
7. while: No. if: Yes. ⇒ push(2,6,⊥)
8. while: Yes. ⇒ report(2,6,7)
   while: No. if: No.
9. while: No. if: Yes. ⇒ push(1,8,⊥)
10. while: Yes. ⇒ report(1,8,9)
    while: No. if: No.

11. *Clean up stack.* report(0,0,10)

Thus we can generate all lcp-intervals very efficiently. But in order to perform a meaningful bottom-up traversal, it is necessary to propagate information from the leaves towards the root. Thus every lcp-interval needs to know about its children when it is "processed" by the algorithm. The following observation helps.

Theorem. Let *top* be the topmost interval on the stack and top_{-1} be the one next to it on the stack. (Hence $top_{-1}.lcp < top.lcp$.) Now assume that $lcptab[i] < top.lcp$, so that *top* will be popped off the stack in the while loop. Then the following holds.

1. If $lcptab[i] \leq top_{-1}.lcp$, then *top* is the child interval of top_{-1}.
2. If $top_{-1}.lcp < lcptab[i] < top.lcp$, then *top* is the child interval of the $lcptab[i]$-interval that contains i.

In both cases we know the parent of *top*. – Conversely, every stack entry will know its childs!

Proof. The following illustrates the two cases:

```
pos   lcptab[pos]
      0123456789...
20      *              top_{-1}.lb=20   ( =parent.lb   if case 1)
21        *----------<
22        *
23        *            top.lb=23        ( =parent.lb   if case 2)
24        .   *------<
25        .   . *
26        .   *
27        .   *
28        .   *------> top.rb=28, report
i     11111222<======  case (1 or 2)
```

Thus we can extend the lcp-interval tuples by a *child list*: The entries will have the form $(lcp, lb, rb, childList)$, where *childList* is the list of its child intervals.

The lists are extended by using an *add* operation. $add([c_1, \ldots, c_k], c)$ appends c to the list $[c_1, \ldots, c_k]$ and returns the result.

In case 1, we add *top* to the child list of top_{-1}, and top_{-1} is popped next. Otherwise (case 2), the while loop is left without assigning a parent for *top*.

The algorithm of [AKO04] for bottom-up traversal with child information is then as follows.

Algorithm 2. (Bottom-up traversal with child information) $lastInterval := \perp$

```
push((0, 0, ⊥, [ ]))
for $i := 1$ to $n$ do
    $lb := i - 1$
    while $lcptab[i] < top.lcp$
        $top.rb := i - 1$
        $lastInterval := pop$
        $process(lastInterval)$    // knows about children!
        $lb := lastInterval.lb$
        if $lcptab[i] \leq top.lcp$ then    // case (1)
            $top.childList := add(top.childList, lastInterval)$
            $lastInterval := \perp$
    if $lcptab[i] > top.lcp$ then
        if $lastInterval \neq \perp$ then    // case (2)
            $push((lcptab[i], lb, \perp, [lastInterval]))$
            $lastInterval := \perp$
        else
            $push((lcptab[i], lb, \perp, [\,]))$
```

Many problems can be solved merely by specifying the function *process*.

For example, the *multiple genome alignment* algorithm of Höhl, Kurtz, and Ohlebusch (Bioinformatics 18(2002)) finds maximal multiple exact matches (multiMEMs) by bottom-up traversal on an (enhanced) suffix array. (The details shall be worked out in the exercises.)

Another example given in [AKO04] is computing the *Ziv-Lempel decomposition* of a string.

9.8 The child table

An optimal top-down traversal requires that we can, for each ℓ-interval, determine its child intervals in constant time. In order to achieve this goal, the suffix array is enhanced with the additional table *childtab*.

childtab contains three values per index: *up*, *down*, and *nextℓIndex*.

For an ℓ-interval $[i..j]$ with ℓ-indices $i_1 < i_2 < \ldots < i_k$, we have $childtab[i].down = i_1$ or $childtab[j+1].up = i_1$. (Or both; the exact details are a bit more complicated, see a lemma below).

Moreover,

$$childtab[i_p].next\ell Index = i_{p+1} \quad \text{for } p = 1, \ldots, k-1 \ .$$

Definitions.

$$\begin{aligned} &childtab[i].up := \\ &\qquad \min\{q \in [0..i-1] \mid \ lcptab[q] > lcptab[i] \ \text{ and } \\ &\qquad\qquad \forall k \in [q+1..i-1] : lcptab[k] \geq lcptab[q] \ \} \\ &childtab[i].down := \\ &\qquad \max\{q \in [i+1..n] \mid \ lcptab[q] > lcptab[i] \ \text{ and } \\ &\qquad\qquad \forall k \in [i+1..q-1] : lcptab[k] > lcptab[q] \ \} \\ &childtab[i].next\ell Index := \\ &\qquad \min\{q \in [i+1..n] \mid \ lcptab[q] = lcptab[i] \ \text{ and } \\ &\qquad\qquad \forall k \in [i+1..q-1] : lcptab[k] > lcptab[i] \ \} \end{aligned}$$

Undefined values ($\min \emptyset$, $\max \emptyset$) are set to $\perp$. (Labels (1.) and (2.) are referred to in proof below.)

Example. (Only partial information is shown in the table.)

i	*suftab*	*lcptab*	*up*	*down*	*nextℓ...*	$S_{suftab[i]}$
0	2	0		2	6	`aaacatat$`
1	3	2				`aacatat$`
2	0	1	1	3	4	`acaaacatat$`
3	4	3				`acatat$`
4	6	1	3	5		`atat$`
5	8	2				`at$`
6	1	0	2	7	8	`caaacatat$`
7	5	2				`catat$`
8	7	0	7	9	10	`tat$`
9	9	1				`t$`
10	10	0	9			`$`

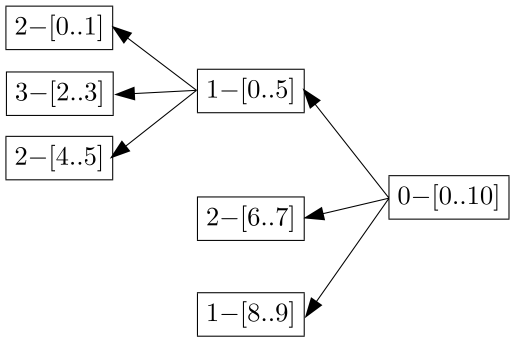
Lemma. Assume we have an ℓ-interval $[i..j]$ with ℓ-indices $i_1 < i_2 < \ldots < i_k$. Then the child intervals of $[i..j]$ are

$$\begin{gathered} [i \mathrel{..} i_1 - 1] \\ [i_1 \mathrel{..} i_2 - 1] \\ \ldots \\ [i_{k-1} \mathrel{..} i_k - 1] \\ [i_k \mathrel{..} j] \,. \end{gathered}$$

Some of these can be singleton intervals.

Example.

i	*suftab*	*lcptab*	*up*	*down*	*nextℓ...*	$S_{suftab[i]}$
0	2	0		2	6	`aaacatat$`
1	3	2				`aacatat$`
2	0	1	1	3	4	`acaaacatat$`
3	4	3				`acatat$`
4	6	1	3	5		`atat$`
5	8	2				`at$`
6	1	0	2	7	8	`caaacatat$`
7	5	2				`catat$`
8	7	0	7	9	10	`tat$`
9	9	1				`t$`
10	10	0	9			`$`



Lemma. The child table can be constructed in linear time using the algorithm for bottom-up traversal with child information.

Proof. Exercise.

Note: [AKO04] actually give two (more direct) algorithms to separately construct the up/down values and the *nextℓIndex* values of the child table.

The child table can be "compressed" so that it uses only one instead of three fields ".up" ".down" and ".*nextℓIndex*" because they contain (to some extent) redundant information. One can then reconstruct the full information by some redirections and case distinctions. We omit the rather tricky details.

Thus the three names are only used for clarity of exposition, and the space requirement for the *childtab* table is 4 bytes per character.

9.9 Top-down traversals

Now we show how the child table can be used to perform a top-down traversal of a (virtual) suffix tree that is actually represented by an enhanced suffix array with *childtab* and *lcptab* information.

We want to retrieve the child intervals of an ℓ-interval $[i..j]$ in constant time. The first step is to find the position of the first ℓ-index in $[i..j]$ (i. e., the minimum of the set $\ell Indices[i.j]$).

The following lemma shows that this is possible with the help of the .up and .down fields of the *childtab*.

Lemma. For every ℓ-interval $[i..j]$ the following holds:

1. We have $i < childtab[j+1].up \leq j$ or $i < childtab[i].down \leq j$.
2. If $i < childtab[j+1].up \leq j$, then $childtab[j+1].up$ stores the first ℓ-index in $[i..j]$.
3. If $i < childtab[i].down \leq j$, then $childtab[i].down$ stores the first ℓ-index in $[i..j]$.

Corollary. The following function $getlcp(i, j)$ returns the lcp-value of an lcp-interval $[i..j]$ in constant time:

```
getlcp (i, j)
if ( $i < childtab[j+1].up \leq j$ )
then
    return $childtab[j+1].up$
else
    return $childtab[i].down$
```

Proof. Let $u := childtab[j+1].up$ and $d := childtab[i].down$.

By definition of the *.up* and*.down* fields, it is clear that $u \leq j$ and $i < d$.

Since $[i..j]$ is an ℓ-interval, we have $lcptab[i] < \ell$ and $lcptab[j+1] < \ell$ and $\forall k \in [i{+}1..j] : lcptab[k] \geq \ell$.

Proof of (1.)
Case 1: $lcptab[i] \geq lcptab[j+1]$
Then $d < j+1$, because otherwise we had $i < j+1 \leq d$ and hence, by definition of *.down*, $lcptab[j+1] \geq lcptab[d] > lcptab[i]$. Thus $i < d \leq j$ and (1) holds.
Case 2: $lcptab[i] < lcptab[j+1]$

Then $i < u$, because otherwise we had $u \le i < j + 1$ and hence, by definition of *.up*, $lcptab[i] \ge lcptab[u] > lcptab[j + 1]$. Thus $i < u \le j + 1$ and again, (1) holds.

For the proof of (2) and (3), let f be the first ℓ-index in $[i..j]$.

Proof of (2.)
Assume $i < u < j$. Thus $lcptab[u] \ge \ell$. We have $u \le f$ because $[i..j]$ has at least one ℓ-index, and every ℓ-index satisfies the conditions for the set in the definition of *.up*. On the other hand, $u \ge f$ because otherwise we had $u < f < j + 1$ and hence, $lcptab[u] < lcptab[f]$ $(= \ell)$ according to the definition of *.up*. Thus $f = u$ as claimed.

Proof of (3.)
Assume $i < d < j + 1$. Thus $lcptab[d] \ge \ell$. We have $d \ge f$ because f satisfies the conditions of the set in the definition of *.down*. On the other hand, we have $d \le f$ because otherwise, $i < f < d$ and hence, $lcptab[f] > lcptab[d]$ $(\ge \ell)$ according to the definition of *.down*. Thus $f = d$ as claimed.

■

Once the first ℓ-index of an ℓ-interval $[i..j]$ has been found, the remanining ℓ-indices $i_2 < i_3 < \ldots < i_k$, where $1 \le k \le \Sigma$, can obtained successively using the *nextℓIndex* fields.

Algorithm.

```
getChildIntervals( $\ell$-[i..j] : lcp-interval )
intervalList = [ ]
if ( $i < childtab[j+1].up \le j$ )
    then $i_1 = childtab[j+1].up$
    else $i_1 = childtab[i].down$
add(intervalList, $(i, i_1 - 1)$)
while ( $childtab[i_1].next\ell Index \ne \perp$ ) do
    $i_2 := childtab[i_1].next\ell Index$
    add(intervalList, $(i_1, i_2 - 1)$)
    $i_1 = i_2$
add(intervalList, $(i_1, j)$)
```

Since $|\Sigma|$ is a constant, algorithm getChildIntervals runs in constant time.

Using getChildIntervals, one can simulate every top-down traversal of a suffix tree on an enhanced suffix array.

For example, one can easily modify the function getChildIntervals to a function *getInterval* that takes as arguments an ℓ-interval $[i..j]$ and a character $a \in \Sigma$ and returns the child interval $[l..r]$ of $[i..j]$ whose suffixes have a character a at position ℓ. (If no such subinterval exists, we return $\perp$.)

Using a top-down traversal, one can search for a pattern P in optimal $|P|$ time, as explained in the next section.

Another application mentioned in [AKO04] is finding all shortest unique substrings. This problem arises e. g. in the design of PCR primers. A substring of S is unique if it occurs only once in S. The shortest unique substring problem is to find all shortest unique substrings of S. For example, acac has only one shortest unique substring, ca. It is easy to see that unique substrings of S correspond to singleton lcp-intervals. Among those, we want to enumerate all with the minimal *lcp* value. This can be accomplished by a breadth-first-search traversal of the lcp-interval tree.

9.10 Searching for substrings in optimal time

Algorithm.

```
answering_decision_queries( P : pattern )
$c := 0$                                                    // current pattern position
$queryFound := true$
$(i, j) := getInterval(0, n, P[c])$
while ( $(i, j) \neq \perp$ and $c < m$ and $queryFound = true$ )
    if ( $i \neq j$ )
    then
        $\ell := getlcp(i, j)$
        $min := \min\{\ell, m\}$
        $queryFound := \big(S[suftab[i]+c\ ..\ suftab[i]+min-1] == P[c..min-1]\big)$
        $c := min$
        $(i, j) := getInterval(i, j, P[c])$
    else
        $queryFound := \big(S[suftab[i]+c\ ..\ suftab[i]+m-1] == P[c..m-1]\big)$
if ( $queryFound$ )
    then report $(i, j)$                                    // the P-interval of suftab
    else report "P not found"
```

The running time of the suffix array based algorithm is $O(|P|)$, the same as for the suffix tree based algorithm.

Enumerative queries can be answered in optimal $O(|P| + z)$ time, where z is the number of occurrences of P. The algorithm is the same, however instead of reporting (i, j), we output $suftab[i], \dots suftab[j]$.