5 Enhanced Suffix Arrays

This exposition by Knut Reinert 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. PhD thesis David Weese: Indices and Applications in High Throughput sequencing

5.1 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.

5.2 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 $\omega = S[i_1..j_1]$.
- A *supermaximal repeat* is a maximal repeat that never occurs as a substring in any other maximal repeat.

5.3 The basic tables

- ***suftab***: The suffix table. An array of integers in the range 0 to $n-1$, specifying the lexicographic ordering of the n suffixes of the string S. Requires $4n$ bytes.
- ***sufinv***: The inverse of the suffix table. An array of integers in the range 0 to $n-1$ 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*: $lcptab[i] := lcp(S_{suftab[i]}, S_{suftab[i-1]})$ for $0 < i < n$ and $lcptab[i] := -1$ for $i \in \{0, n\}$. 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.

5.4 Lcp-intervals

Definition. Let $i, j \in \mathbb{N}$ with $[i..j) \subseteq [0..n)$ and $j - i \geq 2$. Then $[i..j)$ is an *lcp-interval of lcp-value* ℓ iff:

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

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

For completeness, we also define $[i..i+1)$ to be a (singleton) ℓ-interval with $\ell = |S_{suftab[i]}|$ and $i \in [0..n)$.

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

$[i..j)$ is the *ω-interval*, where ω is the longest common prefix of $S_{suftab[i]}, \ldots, S_{suftab[j-1]}$.

The example below shows a string S and the corresponding tables *suftab* and *lcptab*. What are the lcp-intervals?

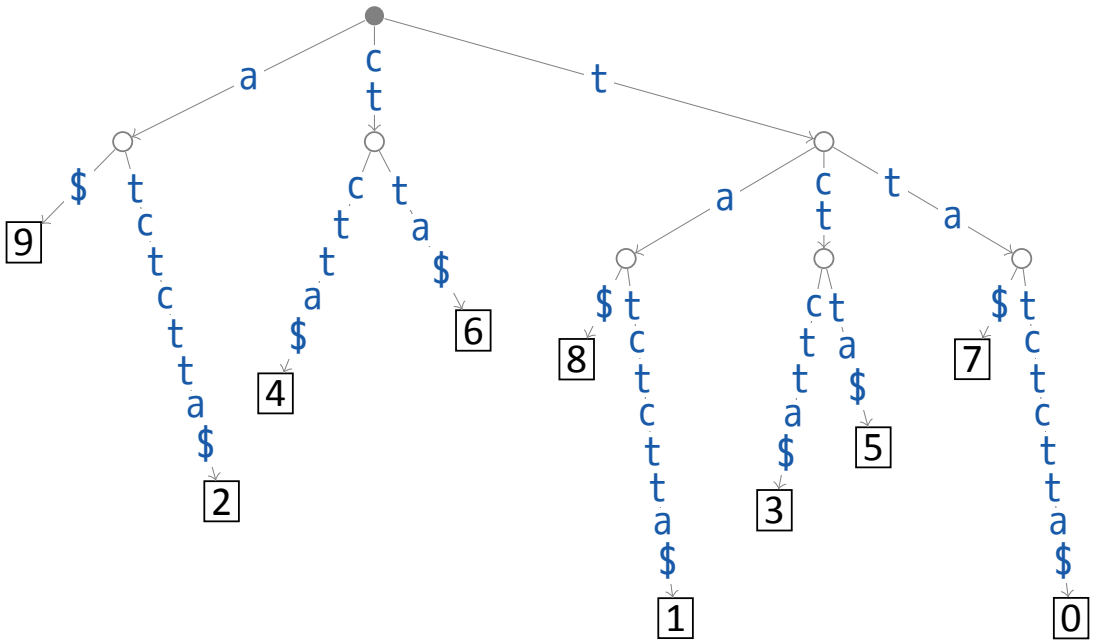
	0	1	2	3	4	5	6	7	8	9
$s =$	t	t	a	t	c	t	c	t	t	a

	0	1	2	3	4	5	6	7	8	9	
suftab =	9	2	4	6	8	1	3	5	7	0	
lcp =	-1	1	0	2	0	2	1	3	1	3	-1

```
a  a  c  c  t  t  t  t  t  t
   t  t  t  a  a  c  c  t  t
   c  c  t     t  t  t  a  a
   t  t  a     c  c  t     t
   c  t        t  t  a     c
   t  a        c  t        t
   t           t  a        c
   a           t           t
               a           t
                           a
```

Here is the corresponding suffix tree[1]:

[1]The sentinel `$` was appended in the example to separate whole suffixes from inner nodes.



Lemma 1 (node-interval-duality)**.** *For every suffix tree node v in a suffix tree ST of a string s, there is an ω-interval $[i..j)$ in the suffix array $suftab$ and vice versa. If v is an inner node it holds $\omega = concat(v)$ and otherwise $\omega\$ = concat(v)$.*

Proof:
For each singleton ω-interval holds: ω is a suffix of s and hence there is a leaf v in the suffix tree of s with $concat(v) = \omega\$$. Analogously, for each leaf there is an ω-interval. Now, we consider non-singleton ω-intervals and their corresponding inner nodes v.

($\rightarrow$) For every inner node v of the suffix tree $ST(s)$ there is a set of suffixes of s beginning with $concat(v)$. Let $S(v)$ denote this set. The suffixes in $S(v)$ correspond to the leaves in the subtree rooted at v. As an inner node, v has at least two outgoing edges beginning with $a, b \in \Sigma \cup \{\$\}$ and $a \neq b$. Therefore, there are at least two suffixes in $S(v)$ that begin with $concat(v)\alpha$ and $concat(v)\beta$, where $\alpha, \beta \in \Sigma^0 \cup \Sigma^1$ and $\alpha \neq \beta$.

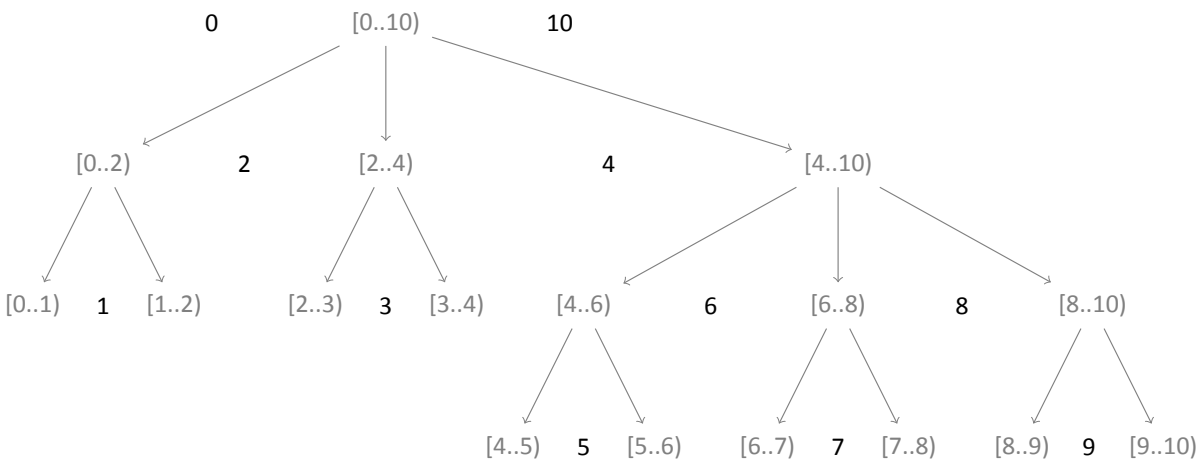
Thus the longest-common prefix of $S(v)$ is $concat(v)$. As all suffixes beginning with a common prefix are stored in a contiguous interval, there are i and j, such that $suftab[i..j)$ stores the begin positions of all suffixes in $S(v)$. Suffixes that are not included in $S(v)$ are not in the subtree rooted at v and do not begin with $concat(v)$. Therefore $lcp[i] < \ell$ and $lcp[j] < \ell$ and $[i..j)$ is an ω-interval with $\omega = concat(v)$.

($\leftarrow$) Let $[i..j)$ be a non-singleton ω-interval and $T(\omega)$ denote the set of suffixes of s beginning with ω. By definition $\{s_{suftab[i]}, \ldots, s_{suftab[j-1]}\} \subseteq T(\omega)$. As the ω-interval is maximal it follows $T(\omega) = \{s_{suftab[i]}, \ldots, s_{suftab[j-1]}\}$. By property 4 exists a k such that $s_{suftab[k-1]}$ and $s_{suftab[k]}$ begin with $\omega\alpha$ and $\omega\beta$, where $\alpha, \beta \in \Sigma^0 \cup \Sigma^1$ and $\alpha \neq \beta$. As a consequence, the lowest common ancestor (lca) of the leaves representing these suffixes is a branching node v with $concat(v) = \omega$ and $S(v) = T(\omega)$.

Corollary 2. *Every suffix tree node v can be identified by an lcp-interval $[i..j)$ and both represent the same set of suffixes $S(v) = s_{suftab[i]}, s_{suftab[i+1]}, \ldots, s_{suftab[j-1]}$.*

Let v be an inner suffix tree node with children $w_1, \ldots, w_m$. W.l.o.g. let $concat(w_1) <_{lex} concat(w_2) <_{lex} \ldots <_{lex} concat(w_m)$. Obviously the sets $S(w_1), S(w_2), \ldots, S(w_m)$ form a partition of the set $S(v)$. The lcp-intervals of the children (child intervals) are subintervals that form a partition $[l_0..l_1), [l_1..l_2), \ldots, [l_{m-1}..l_m)$ of the ℓ-interval $[i..j)$ of v, where $l_0 = i$ and $l_m = j$. The length of the longest-common prefix of suffixes from different child subtrees is $\ell = |concat(v)|$, whereas the lcp-length of suffixes from the same subtree is greater than ℓ. Thus for $x \in [i, j)$ it holds $lcp[x] = \ell$ only if $x \in \{l_1, \ldots, l_{m-1}\}$ and $lcp[x] > \ell$, otherwise. The indices $l_1, \ldots, l_{m-1}$ uniquely define the partition into subintervals and are called *ℓ-indices* of the lcp-interval $[i..j)$. The set $\{l_1, \ldots, l_{m-1}\}$ is denoted by ℓ-indices(i, j).

And again the lcp-intervals depicted as tree together with the ℓ-indices.



5.5 The child table

The parent-child relationship of lcp-intervals corresponds to the parent-child relationship of suffix tree nodes and constitutes the so-called lcp-interval tree.

The child table is a linked list of ℓ-indices and stores for each ℓ-index so-called *up*, *down*, and *nextl* values. It can be represented as 3 subtables which are strings of length $n+1$ over the alphabet $[0..n]$.

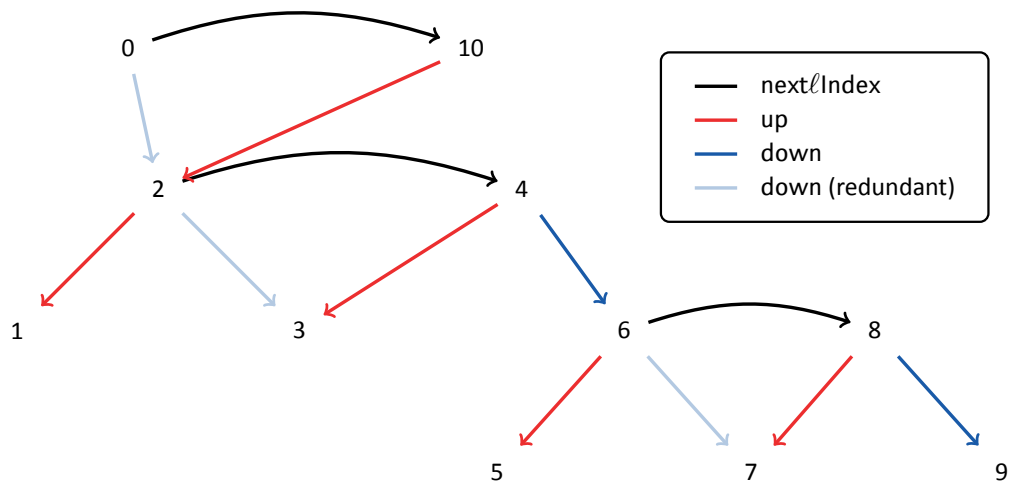
For $l_k \in \ell\text{-indices}(i, j)$, $nextl(l_k)$, if existent, is the next greater ℓ-index l_{k+1} in the set $\ell\text{-indices}(i, j)$. $up(l_k)$ and $down(l_k)$, if existent, are the smallest ℓ-indices in the sets $\ell\text{-indices}(l_{k-1}, l_k)$ and $\ell\text{-indices}(l_k, l_{k+1})$. For an arbitrary ℓ-index i, the values *up*, *down*, and *nextl* can formally be defined as follows:

$$up(i) = \min\{q \in [0..i) \mid lcp[q] > lcp[i] \ \wedge \ \forall_{k\in(q..i)} lcp[k] \geq lcp[q]\}\,, \tag{5.1}$$

$$down(i) = \max\{q \in (i..n] \mid lcp[q] > lcp[i] \ \wedge \ \forall_{k\in(i..q)} lcp[k] > lcp[q]\}\,, \tag{5.2}$$

$$nextl(i) = \min\{q \in (i..n] \mid lcp[q] = lcp[i] \ \wedge \ \forall_{k\in(i..q)} lcp[k] > lcp[i]\}\,. \tag{5.3}$$

Navigating in a suffix array.



The child table.

i	u.	d.	n.	cld
0		2	10	10
1				1
2	1	3	4	4
3				3
4	3	6		6
5				5
6	5	7	8	8
7				7
8	7	9		9
9				2
10	2			

Abouelhoda et al. proposed an easy and elegant way to reduce the memory consumption of the child table by two-thirds. Instead of using 3 strings of length $n+1$, they merge the 3 subtables into a single string *cld* of length n over the alphabet $[0..n]$. Their method benefits from the following two observations:

Observation 3. Each defined value $nextl(i)$ can be stored at $cld[i]$. For the last ℓ-index i in every lcp-interval $nextl(i)$ is undefined and the entry $cld[i]$ can be used to store $down(i)$ instead. For all other ℓ-indices the *down*-value equals the *up*-value of its successor and needs not to be stored explicitly.

Observation 4. For every ℓ-index i that has a defined *up*-value, $i-1$ is an ℓ'-index with undefined values for *down* and *nextl*. Thus the *up*-value can be stored at $cld[i-1]$.

Proof: Let $q := cld[i].up$ be the *up*-value of i. By the formal definition holds $lcp[q] > lcp[i]$ and $lcp[k] \geq lcp[q] > lcp[i]$ for every $k \in (q..i)$. It especially holds $lcp[i-1] > lcp[i]$ and thus the values $down(i-1)$ and $nextl(i-1)$ are undefined as for all $q' \geq i+1$ and $k' = i+1$ the necessary condition $lcp[k'] > lcp[q'] \geq lcp[i]$ is violated.

Abouelhoda et al. showed how to retrieve the original three values from the merged child table *cld*:

$$up[i] = \begin{cases} cld[i-1], & \text{if } i > 0 \text{ and } lcp[i-1] > lcp[i] \\ \perp, & \text{else,} \end{cases} \tag{5.4}$$

$$down[i] = \begin{cases} cld[i], & \text{if } lcp[cld[i]] > lcp[i] \\ \perp, & \text{else,} \end{cases} \tag{5.5}$$

$$nextl[i] = \begin{cases} cld[i], & \text{if } lcp[cld[i]] = lcp[i] \\ \perp, & \text{else .} \end{cases} \tag{5.6}$$

For each non-singleton lcp-interval $[i..j)$ either $down[i]$ or $up[j]$ is defined and equals the first ℓ-index l_1. The other ℓ-indices can be determined by $l_{k+1} = nextl[l_k]$.

The following figure shows the compressed child table.

i	u.	d.	n.	cld
0		2	10	10
1				1
2	1	3	4	4
3				3
4	3	6		6
5				5
6	5	7	8	8
7				7
8	7	9		9
9				2
10	2			

5.6 Computing the child table

The lcp table alone can be used to traverse the inner nodes of the suffix tree in a bottom-up fashion (exercise). In linear time the algorithm outputs all ℓ-intervals $[lb..rb)$ that correspond to suffix tree nodes visited in a postorder depth-first search (DFS).

Therefore it scans the lcp-table and maintains a stack of growing lcp-table intervals and their minimal lcp value ℓ.

Abouelhoda et al. propose two modifications of this bottom-up algorithm to construct the *up* and *down* values and the *nextl* values. In the next algorithm (proposed by Weese) we show the combination of both algorithms to directly construct the child table. Instead of ℓ-values and interval boundaries used in the bottom-up traversal, the stack in this algorithm only stores ℓ-indices, i. e. each ℓ-interval is represented by a run of its ℓ-indices.

According to the space-saving trick described above, *up* values for ℓ-indices i are stored at position $i-1$ (line 11) and *down* values are stored only for the last of all ℓ-indices of each interval (lines 8).

```
(1)  // constructChildTable(lcp)
(2)  input is the lcp table, output the child table cld;
(3)  n ←| lcp | −1;
(4)  S ← 0; last ← −1;
(5)  for i = 1 to n do
(6)      while lcp[i] < lcp[top(S)] do
(7)          last ← pop(S);
(8)          if lcp[i] < lcp[top(S)] ∧ lcp[top(S)] ≠ lcp[last] then cld[top(S)] = last; fi
(9)      od
(10)     if last ≠ −1 then
(11)                 cld[i − 1] = last;
(12)                 last = −1;
(13)     fi
(14)     if lcp[top(S)] = lcp[i] then cld[top(S)] = i; fi
(15)     push(i, S);
(16) od
(17) return cld;
```

The condition in line 8 is true, iff the two ℓ-values on the top of the stack are different from each other and

greater than the current ℓ-value $lcp[i]$. In this case, both elements will be removed and are ℓ-indices from an lcp-interval and its last-child interval. As ℓ-indices of the same interval are stored as an ascending run, the topmost stack entry (*last*) is the first ℓ-index in the last-child interval and the second topmost stack entry (now $top(S)$) is the last ℓ-index in the parent interval and the left border of the last-child interval. Hence, *last* is its *down* value and needs to be stored.

In line 10, *last* is the last ℓ-index removed by the current ℓ-index i or equals -1 if none was removed. In the first case, the last removed ℓ index is the first ℓ-index in the child interval left of i. Hence, *last* is the *up* value of i and is stored at position $i - 1$.

nextl values are computed similarly. If after the removal of all greater ℓ-values the topmost ℓ-value equals the current one, the topmost ℓ-index is directly preceding i and its *nextl* value is set accordingly in line 14.

With the enhanced suffix array consisting of suffix array, lcp table, and child table, we are now able to top-down traverse the nodes of the suffix tree of a text which actually is a node traversal of the corresponding lcp-interval tree.

We start with a *top-down iterator* and functions to go to the root node, to go down to the leftmost child, and to go right to the next sibling of the currently visited node, where the children are lexicographically ordered by their edge labels from left to right.

The top-down iterator maintains the values *lb* and *rb*, the lcp-interval boundaries of the currently visited node, and *parentRb* the right boundary of the parent node. It starts in the root of the lcp-interval tree which is the interval $[lb..rb) = [0..n)$. As the root node has no right sibling, *parentRb* is set to *rb*. The intervals in the lcp-interval tree are distinct from each other and two iterators can be compared by comparing their boundary pairs. For leaf nodes hold $rb - lb = 1$.

When moving the iterator to the first child of the current node, the left boundary remains the same whereas the smallest ℓ-index in ℓ-indices(lb, rb) becomes the new right boundary *rb* and *parentRb* the former value of *rb*.

If the current node is not the last child of its parent ($rb \neq parentRb$), the smallest ℓ-index in ℓ-indices(lb, rb) is the *up* value of *rb* stored at $cld[rb - 1]$, otherwise it is the *down* value of *lb* stored only in this case at $cld[lb]$.

Moving the iterator to the next sibling is possible iff $rb \neq parentRb$, i. e. for all but the last sibling. Then *lb* becomes the former *rb* and *rb* becomes, if existent, its next ℓ-index or *parentRb*, otherwise.

```
// goRoot(iter)
input is an iterator;
iter.lb ← 0;
iter.rb ← n;
iter.parentRb ← n + 1;
```

```
// isNextl(iter)
input is l-index i;
j ← cld[i];
if i < j ∧ lcp[i] = lcp[j] then return TRUE; fi
return FALSE;
```

```
// goDown(iter)
input is an iterator;
if iter.rb − iter.lb ≤ 1 then return FALSE; fi
if iter.rb ≠ iter.parentRb then
                    iter.parentRb ← iter.rb;
                    //get up value of right boundary
                    iter.rb ← cld[iter.rb − 1];
                else
                    //get down value of left boundary
                    iter.rb ← cld[iter.lb];
fi
return TRUE;
```

```
(1)  // goRight(iter)
(2)  input is an iterator;
(3)  if iter.rb = iter.parentRb ∨ iter.parentRb = n + 1 then return FALSE; fi
(4)  iter.lb ← iter.rb;
(5)  if isNextl(iter.rb) then
(6)                  iter.rb ← cld[iter.rb];
(7)                  //iter.rb has a succeeding ℓ-index
(8)              else
(9)                  //iter.lb is the last ℓ-index
(10)                 iter.rb ← iter.parentRb;
(11) fi
(12) return TRUE;
```

If only the 3 tables of the enhanced suffix array are given, it is not possible to move a top-down iterator upwards the tree in $O(1)$ time.

It is however possible to recover the state the iterator had in the parent node by manually maintaing a stack of top-down iterator copies.

The stack stores the lcp-interval boundaries of nodes on the path to the root. The *goRoot* and *goDown* functions can be adapted to clear the stack and to push the current interval boundary pair (lb, rb) onto the stack before going down.

The new function *goUp* simply replaces (lb, rb) by the topmost pair and removes it from stack. We also need to adapt *parentRb*, which is set to the new topmost *rb* value on stack.

5.7 Accessing the suffix tree

All of the suffix tree iterators described above store a pair of boundaries of the current lcp-interval $[lb..rb)$. We are not only able to traverse the lcp-interval tree but also to access all information about the corresponding suffix tree, given only the suffix array, the lcp table, and the node boundary pairs.

In the following, we are going to determine the concatenation string of the current node, i. e. the concatenation of characters on the path from the root node to the node the iterator points at. If v is the suffix tree node that corresponds to the lcp-interval $[lb..rb)$, the concatenation string without sentinel equals the longest-common prefix ω of the ℓ-interval $[lb..rb)$.

Thus, ω especially is the ℓ-prefix of the lexicographically smallest suffix of the ℓ-interval and it holds $\omega = s_{suftab[lb]}[0..\ell)$. It remains to determine ℓ.

If $rb - lb = 1$, v is a leaf and ℓ equals the suffix length $\left|s_{suftab[lb]}\right| = n - suftab[lb]$. For $rb - lb > 1$, v is an inner node and ℓ is the lcp value of any, especially the smallest index in ℓ-indices(lb, rb).

In line 3 of the function *repLength*, the child table entry at position *iter.lb* is read. It either contains a *nextl*-value if the current node is the first or an inner sibling or a *down*-value if the node is the last sibling. In the latter case, the entry is less than *iter.rb*. If it is greater than or equal to *iter.rb*, it contains a *nextl*-value, the current node is not the last sibling and the right boundary is an ℓ-index with an *up*-value at position $iter.rb - 1$. The concatenation string of a node is returned by the function.

```
(1)  // repLength(iter)
(2)  input is an interator;
(3)  if iter.rb − iter.lb = 1 then return |s| − suftab[iter.lb]; fi
(4)  i ← cld[iter.lb];
(5)  // try to get down value of left boundary
(6)  if iter.rb ≤ i then //was it the nextlIndex value?
(7)                  i ← cld[iter.rb − 1];
(8)                  //get up value of right boundary
(9)  fi
(10) return lcp[i]; //value of the ℓ-interval [iter.lb..iter.rb).
```

```
(1) // representative(iter)
(2) input is an interator;
(3) i ← suftab[iter.lb];
(4) return s[i..i + repLength(iter));
```

To only determine the label of the edge from the parent to the current node, we need to compute the ℓ-value of the parent interval of $[lb..rb)$ in the interval tree.

Assume $[lb..rb)$ is not the root node and $[lb'..rb')$ its parent interval. Above we have seen that the child interval boundaries are contained in the set ℓ-indices$(lb', rb') \cup \{lb', rb'\}$ and at least one boundary of every interval is an ℓ-index of the ℓ-interval $[lb'..rb')$. Thus, either both boundaries are ℓ-indices or only one is an ℓ-index and then the other's lcp value must be less than ℓ. In either case the maximal lcp value of the boundaries equals ℓ. For the case that $[lb..rb) = [0..n)$ is the root interval, we set $\ell = 0$ as the root has no parent edge. Thus, it holds $\ell = \max\{lcp[lb], lcp[rb], 0\}$ and the parent edge label is the suffix of the concatenation string starting at position ℓ.

```
(1) // parentEdgeLabel(iter)
(2) input is an interator;
(3) l ← max{lcp[iter.lb], lcp[iter.rb, 0};
(4) t ← representative(iter);
(5) return t[l.. | t |);
```

The occurrences of a string t in the text s are the start positions of suffixes of s beginning with t. Hence, if t is the concatenation string of a suffix tree node, its occurrences can be determined by traversing the leaves in the node's subtree. Given the enhanced suffix array, the set of suffix start positions can directly be obtained, as for a node $[lb..rb)$ the substring $suftab[lb..rb)$ contains the start positions of its concatenation string. The below algorithm shows the corresponding pseudo-code.

```
(1) // getOccurrences(iter)
(2) input is an interator;
(3) for i ← iter.lb to iter.rb − 1 do
(4)     print"occurrence at position"suftab[i];
```

5.8 Searching maximal repeats

Given a text s. A string α is called a *maximal repeat*, iff there is at least one maximal repeated pair $(p_1, p_2, |\alpha|)$ with $\alpha = s[p_1..p_1 + |\alpha|) = s[p_2..p_2 + |\alpha|)$. For example, the text $s =$ `xabcyabcwabcyz` contains the two maximal repeats `abc`, with maximal repeated pairs $(1, 5, 3)$ and $(5, 9, 3)$, and `abcy` with the only maximal repeated pair $(1, 9, 4)$ depicted below.

$$s = \begin{cases} \texttt{XabcYabcWABCYZ} \\ \texttt{XABCYabcWabcYZ} \\ \texttt{XabcyABCWabcyZ} \end{cases}$$

The the contains 3 maximal repeated pairs of 2 repeats (a) and only one supermaximal repeat `XABCYabcWabcYZ`.

For a given text s and a minimum length n_0, the *maximal repeat problem* is to find all maximal repeats α with $|\alpha| \geq n_0$ and all corresponding maximal pairs. According to the observation above, the right-maximality is given for every inner node v, i. e. every pair of suffixes s_{p_1} and s_{p_2} from different subtrees of v form a right maximal repeated pair $(p_1, p_2, |concat(v)|)$. The pair is also left maximal, if p_1 or p_2 equals 0 or the characters left of the suffixes differ. If at least one such pair exists for a node v, *concat(v)* is a maximal repeat.

Fundamental to the algorithm is to traverse the suffix tree from bottom up and for every tree node to partition the sets of suffixes in the subtree according to their preceding character. To well-define the preceding character, we define \$ to precede the suffix starting at position 0. For a tree node $[i..j)$ and a character $x \in \Sigma \cup \{\$\}$,

the partition of start positions of suffixes preceded by character x in the subtree of $[i..j)$ is:

$$P_{[i..j)}(x) = \begin{cases} \left\{ 0 \;\middle|\; 0 \in suftab[i..j) \right\}, & \text{if } x = \$ \\ \left\{ p \in suftab[i..j) \;\middle|\; p \neq 0 \wedge s[p-1] = x \right\}, & \text{else.} \end{cases} \tag{5.7}$$

Let v be an inner suffix tree node that corresponds to the lcp-interval $[i..j)$ with child intervals $[l_0..l_1)$, $[l_1..l_2)$, $\ldots$, $[l_{m-1}..l_m)$, where $l_0 = i$ and $l_m = j$ holds. The set of maximal repeated pairs for *concat(v)* is the union of Cartesian products of partitions from different subtrees (right-maximality) and different preceding characters (left-maximality):

$$\mathcal{R}_{[i..j)} = \bigcup_{\substack{a,b \in [0..m), \\ a<b}} \bigcup_{\substack{x,y \in \Sigma \cup \{\$\}, \\ x \neq y}} P_{[l_a..l_{a+1})}(x) \times P_{[l_b..l_{b+1})}(y) \times \{|concat(v)|\} . \tag{5.8}$$

If the set $\mathcal{R}_{[i..j)}$ is empty, the concatenation string *concat(v)* is not a maximal repeat. Otherwise, it contains all maximal repeated pairs of the maximal repeat *concat(v)*.

The algorithm computes the sets $P_{[i..j)}(x)$ for every node $[i..j)$ in the lcp-interval tree from bottom up. It begins in the leaves $[k..k+1)$, where $P_{[k..k+1)}(x)$ is empty for all but one character $x \in \Sigma \cup \{\$\}$. It is non-empty and equals the singleton $\{k\}$ for the character that precedes the suffix s_k.

Whenever a child node $[l_b..l_{b+1})$ is visited the last time during the postorder DFS, its sets $P_{[l_b..l_{b+1})}(\cdot)$ are joined to the sets of its parent node $[i..j)$. At the moment, between leaving the child node and appending its sets, the parent $[i..j)$ stores the union of sets of all left siblings of $[l_0..l_1)$, $\ldots$, $[l_{b-1}..l_b)$ which equals $P_{[l_0..l_b)}(\cdot)$. The Cartesian products of the sets $P_{[l_0..l_b)}(x)$ and $P_{[l_b..l_{b+1})}(y)$, with $x \neq y$, constitute maximal pairs and are output for every child $[l_b..l_{b+1})$, with $b \in [1..m)$. It becomes clear that the algorithm is correct, after equivalently rewriting equation 5.8:

$$\mathcal{R}_{[i..j)} = \bigcup_{b \in [1..m)} \bigcup_{\substack{x,y \in \Sigma \cup \{\$\}, \\ x \neq y}} P_{[l_0..l_b)}(x) \times P_{[l_b..l_{b+1})}(y) \times \{|concat(v)|\} . \tag{5.9}$$

The time required for enumerating these maximal repeated pairs is proportional to the number of pairs.

Theorem 5. *The time to enumerate all maximal repeats of a text of length n is $O(n|\Sigma|)$. If the text contains overall k maximal repeated pairs of minimal length n_0, the overall time to output them is $O(n|\Sigma| + k)$.*