Referát

Vyhľadávanie v texte, Tibor Hegedüs

11. júna 2006

Michal Klempai, Veronika Vlnková

Replacing suffix trees with enhanced suffix arrays

Abstract

The suffix tree is one of the most important data structures in string processing and comparative genomics. However, the space consumption of the suffix tree is a bottleneck in large scale applications such as genome analysis. In this article, we will overcome this obstacle. We will show how every algorithm that uses a suffix tree as 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. The new algorithms are not only more space efficient than previous ones, but they are also faster and easier to implement.

Introduction

The suffix tree is undoubtedly one of the most important data structures in string processing. The suffix tree of a sequence S is an index structure that can be computed and stored in O(n) time and space, where $n = |S|$. Once constructed, it can be used to efficiently solve a "myriad" of string processing problems. These applications can be classified into the following kinds of tree traversals:

- a bottom-up traversal of the complete suffix tree,
- a top-down traversal of a subtree of the suffix tree,
- a traversal of the suffix tree using suffix links.

Table 1
The suffix tree applications from [15] and the kinds of traversals they require

Application	Type of tree traversal		
	Bottom-up	Top-down	Suffix-links
Supermaximal repeats	√		
Maximal repeats	√		
Maximal repeated pairs	√		
Longest common substring	√		
All-pairs suffix-prefix matching	√		
Ziv–Lempel decomposition	√		
Common substrings of multiple strings	√	√	
Exact string matching		√	
Exact set matching		√	
Matching statistics		√	√
Construction of DAWGs		√	√

While suffix trees play a prominent role in algorithmics, they are not as widespread in actual implementations of software tools as one should expect. There are two major reasons for this:

(i) Space consumption of a suffix tree is quite large.
(ii) And in most applications, there is a significant loss of efficiency on cached processor architectures.

Basic notions

Let Σ be a finite ordered *alphabet*. Σ^* is the *set of all strings over* Σ. We use Σ^+ to denote the set $\Sigma^* \setminus\{\varepsilon\}$ of non-empty strings. Let S be a string of length $|S| = n$ over Σ. We suppose that the size of the alphabet is a constant, and that $n< 2^{32}$ (an integer in the range $[0, n]$ can be stored in 4 bytes). Special symbol \$ is an element of Σ (which is larger then all other elements) but does not occur in S. $S[i]$ denotes the *character at position* i in S, for $0 \le i < n$. For $i \le j$, $S[i..j]$ denotes the *substring* S starting with the character at position i and ending with the character at position j. The substring $S[i..j]$ is also denoted by the *pair (i, j) of positions*.

A *suffix tree* for the string S is a rooted directed tree with exactly n+1 leaves numbered 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S\$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i , the concatenation of the edge-labels on the path from the root to leaf i exactly spells out
the string S_i, where $S_i = S[i..n - 1]\$$ denotes the i th nonempty suffix of the string S\$, $0 \le i \le n$. Fig. 1 shows the suffix tree for the string $S = acaaacatat$.

The *suffix array* suftab of the string S is an array of integers in the range 0 to n, specifying the lexicographic ordering of the n + 1 suffixes of the string S\$. That is, $S_{\text{suftab}[0]}, S_{\text{suftab}[1]}, \ldots, S_{\text{suftab}[n]}$ is the sequence of suffixes of S\$ in ascending lexicographic order (the suffix array requires 4n bytes).

The *inverse suffix array* suftab^{-1} is a table of size n+1 such that $\text{suftab}^{-1}[\text{suftab}[q]] = q$ for any $0 \le q \le n$. suftab^{-1} can be computed in linear time from the suffix array and needs 4n bytes.

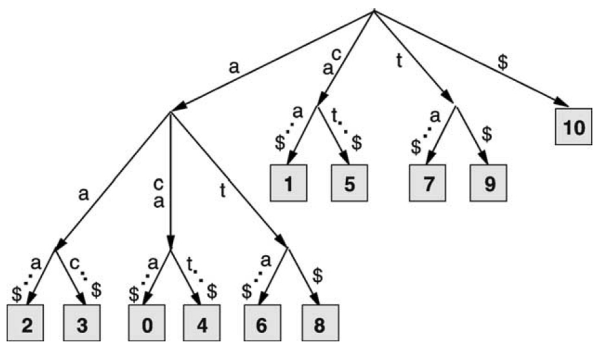


Fig. 1. The suffix tree for $S = acaaacatat$.

The table bwttab contains the *Burrows and Wheeler* transformation [6] known from data compression. It is a table of size $n + 1$ such that for every i, $0 \leq i \leq n$, $\text{bwttab}[i] = S[\text{suftab}[i] - 1]$ if $\text{suftab}[i] \neq 0$. $\text{bwttab}[i]$ is undefined if $\text{suftab}[i] = 0$. The table bwttab is stored in n bytes and constructed in one scan over the suffix array in $O(n)$ time.

The *lcp-table* lcptab is an array of integers in the range 0 to n. We define $\text{lcptab}[0] = 0$ and $\text{lcptab}[i]$ is the length of the longest common prefix of $S_{\text{suftab}[i-1]}$ and $S_{\text{suftab}[i]}$, for $1 \leq i \leq n$. Since $S_{\text{suftab}[n]} = \$$, we always have $\text{lcptab}[n] = 0$. The lcp-table can be computed as a by-product during the construction of the suffix array, or alternatively, in linear time from the suffix array [20]. The lcp-table requires $4n$ bytes in the worst case.

Algorithms based on lcp-intervals

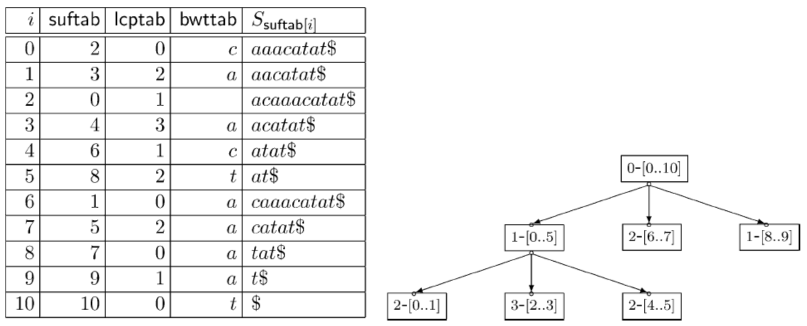
A pair of substrings $R = ((i1, j1), (i2, j2))$ is a *repeated pair* if and only if $(i1, j1) \neq (i2, j2)$ and $S[i1..j1] = S[i2..j2]$. The length of R is $j1 - i1 + 1$. A repeated pair $((i1, j1), (i2, j2))$ is called *left maximal* if $S[i1 - 1] \neq S[i2 - 1]$ and *right maximal* if $S[j1 + 1] \neq S[j2 + 1]$. A repeated pair is called *maximal* if it is left and right maximal. A substring ω of S is a (*maximal*) *repeat* if there is a (maximal) repeated pair $((i1, j1), (i2, j2))$ such that $\omega = S[i1..j1]$. A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat.

The lcp-intervals

Definition: *An interval $[i..j]$, $0 \leq i < j \leq n$, is an lcp-interval of lcp-value l if*

1. $\text{lcptab}[i] < l$,
2. $\text{lcptab}[k] \geq l$ for all k with $i + 1 \leq k \leq j$,
3. $\text{lcptab}[k] = l$ for at least one k with $i + 1 \leq k \leq j$,
4. $\text{lcptab}[j + 1] < l$.

We will also use the shorthand l-interval (or even l-$[i..j]$) for an lcp-interval $[i..j]$ of lcpvalue. Every index k, $i + 1 \leq k \leq j$, with $\text{lcptab}[k] = l$ is called l-index. The set of all l-indices of an l-interval $[i..j]$ will be denoted by *lIndices(i, j)*. If $[i..j]$ is an l-interval such that $\omega = S[\text{suftab}[i]..\text{suftab}[i] + l - 1]$ is the longest common prefix of the suffixes $S_{\text{suftab}[i]}, S_{\text{suftab}[i+1]}, \ldots, S_{\text{suftab}[j]}$, then $[i..j]$ is called the ω-interval.



i	suftab	lcptab	bwttab	$S_{\text{suftab}[i]}$
0	2	0	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	$\$$

Fig. 2. The enhanced suffix array of the string $S = acaaacatat$ and its lcp-interval tree.

A new algorithm for finding supermaximal repeats

Definition: An l-interval $[i..j]$ is called a *local maximum* in the lcp-table if lcptab$[k] = l$ for all $i + 1 \leq k \leq j$. For instance, in the lcp-table of Fig. 2, the local maxima are the intervals [0..1], [2..3], [4..5], [6..7], and [8..9].

Lemma: *A string ω is a supermaximal repeat if and only if there is an l-interval* $[i..j]$ *such that*

- $[i..j]$ *is a local maximum in the lcp-table and* $[i..j]$ *is the ω-interval,*
- *the characters* bwttab$[i]$, bwttab$[i + 1], \ldots,$ bwttab$[j]$ *are pairwise distinct.*

Proof : (If) Since ω is a common prefix of the suffixes $S_{\text{suftab}[i]}, \ldots, S_{\text{suftab}[j]}$ and $i < j$, it is certainly a repeat. The characters $S[\text{suftab}[i]+l], S[\text{suftab}[i+1]+l], \ldots, S[\text{suftab}[j]+l]$ are pairwise distinct because $[i..j]$ is a local maximum in the lcp-table. By the second condition, the characters bwttab$[i]$, bwttab$[i + 1], \ldots,$ bwttab$[j]$ are also pairwise distinct. It follows that ω is a maximal repeat and that there is no repeat in S which contains ω. In other words, ω is a supermaximal repeat.

(Only if) Let ω be a supermaximal repeat of length $|\omega| = l$. Furthermore, suppose that suftab$[i]$, suftab$[i + 1], \ldots,$ suftab$[j]$, $0 \leq i < j \leq n$, are the consecutive entries in suftab such that ω is a common prefix of Ssuftab$[i]$, $S_{\text{suftab}[i+1]}, \ldots, S_{\text{suftab}[j]}$ but neither of $S_{\text{suftab}[i-1]}$ nor of $S_{\text{suftab}[j+1]}$. Because ω is supermaximal, the characters $S[\text{suftab}[i] + l], S[\text{suftab}[i + 1] + l], \ldots, S[\text{suftab}[j] + l]$ are pairwise distinct. Hence lcptab$[k] = l$ for all k with $i + 1 \leq k \leq j$. Furthermore, lcptab$[i] < l$ and lcptab$[j + 1] < l$ hold because otherwise ω would also be a prefix of $S_{\text{suftab}[i-1]}$ or $S_{\text{suftab}[j+1]}$. All in all, $[i..j]$ is a local maximum of the array lcptab and $[i..j]$ is the ω-interval. Finally, the characters bwttab$[i]$, bwttab$[i + 1], \ldots,$ bwttab$[j]$ are pairwise distinct because ω is supermaximal. □

The preceding lemma does not only imply that the number of supermaximal repeats is smaller than n, but it also suggests a simple linear time algorithm to compute all supermaximal repeats of a string S: Find all local maxima in the lcp-table of S. For every local maximum $[i..j]$ check whether bwttab$[i]$, bwttab$[i + 1], \ldots,$ bwttab$[j]$ are pairwise distinct characters. If so, report the string $S[\text{suftab}[i]..\text{suftab}[i] + \text{lcptab}[i] - 1]$ as supermaximal repeat.

Computation of maximal unique matches

Next, we tackle a problem that has its origin in genome comparisons. Nowadays, the DNA sequences of entire genomes are being determined at a rapid rate. When the genomic DNA sequences of closely related organisms become available, one of the first questions researchers ask is how the genomes align. The software tool [8] has been developed to efficiently align two sufficiently similar genomic DNA sequences. In the first phase of its underlying algorithm, a maximal unique match (*MUM*) decomposition of two genomes S_1 and S_2 is computed. Using the suffix tree of $S_1\#S_2$, *MUM*s can be computed in $O(n)$ time and space, where $n = |S_1\#S_2|$ and # is a symbol neither occurring in S_1 nor in S_2. However, the space consumption of the suffix tree has been identified to be a major problem when comparing large genomes. We will solve this problem by using the suffix array enhanced with the lcp-table.

Definition: Given two sequences S_1 and S_2, a *MUM* is a sequence that occurs exactly once in S_1 and once in S_2, and is not contained in any longer such sequence.

Lemma: *Let # be a unique separator symbol not occurring in S_1 and S_2 and let $S = S_1\#S_2$. The string u is a MUM of S_1 and S_2 if and only if u is a supermaximal repeat in S such that*

(1) *there is only one maximal repeated pair $((i1, j1), (i2, j2))$ with $u = S[i1..j1] = S[i2..j2]$,*
(2) $j1 < p < i2$*, where* $p = |S_1|$ *is the position of # in S.*

Proof. (If) It is a consequence of conditions (1) and (2) that *u* occurs exactly once in S_1 and once in S_2. Because the repeated pair $((i1, j1), (i2, j2))$ is maximal, *u* is a *MUM.*

(Only if) If *u* is a *MUM* of the sequences S_1 and S_2, then it occurs exactly once in S_1 (say, $u = S_1[i1..j1]$) and once in S_2 (say, $u = S_2[i2..j2]$), and is not contained in any longer such sequence. Clearly, $((i1, j1), (p+1+i2, p+1+j2))$ is a repeated pair in $S = S_1\$S_2$, where $p = |S_1|$. Because *u* occurs exactly once in S_1 and once in S_2, and is not contained in any longer such sequence, it follows that *u* is a supermaximal repeat in *S* satisfying conditions (1) and (2). □

The first version of software [8] computed *MUM*s in $O(|S|)$ time and space with the help of the suffix tree of $S = S_1\#S_2$. Using an enhanced suffix array, this task can be done more time and space economically as follows: Find all local maxima in the lcptable of $S = S_1\#S_2$. For every local maximum $[i..j]$ check whether $i + 1 = j$, bwttab$[i]$ = bwttab$[j]$, and suftab$[i] < p <$ suftab$[j]$. If so, report S[suftab$[i]$..suftab$[i]$+lcptab$[i]-1$] as *MUM.*

The lcp-interval tree of a suffix array

In [20] was presented a linear time algorithm to simulate the bottom-up traversal of a suffix tree with a suffix array and its lcp-information. It computes all lcp-intervals of the lcptable with the help of a stack. The elements on the stack are lcp-intervals represented by tuples <*lcp, lb, rb*>, where *lcp* is the lcp-value of the interval, *lb* is its left boundary, and *rb* is its right boundary. In Algorithm 1, *push* (pushes an element onto the stack) and *pop* (pops an element from the stack and returns that element) are the usual stack operations, while *top* provides a pointer to the topmost element of the stack. Furthermore, $\perp$ stands for an undefined value.

Algorithm 1 (Computation of lcp-intervals (adapted from Kasai et al. [20])).

```
1.  push(<0, 0,⊥>)
2.  for i := 1 to n do
3.      lb := i − 1
4.      while lcptab[i] < top.lcp
5.          top.rb := i − 1
6.          interval := pop
7.          report(interval)
8.          lb := interval.lb
9.      if lcptab[i] > top.lcp then
10.         push(<lcptab[i], lb,⊥>)
```

Definition: *An m-interval [l..r] is said to be embedded in an l-interval [i..j] if it is a* subinterval of [*i..j*] (i.e., $i \leq l < r \leq j$) and $m > l^2$. The *l*-interval [*i..j*] is then called the interval *enclosing* [*l..r*]. If [*i..j*] encloses [*l..r*] and there is no interval embedded in [*i..j*] that also encloses [*l..r*], then [*l..r*] is called a *child interval* of [*i..j*].

Theorem: *Consider the for-loop of Algorithm* 1 *for some index i. Let top be the topmost interval on the stack and* top_{-1} *be the interval next to it* (*note that* $top_{-1}.lcp < top.lcp$). *If* lcptab[*i*] < *top.lcp, then before top will be popped from the stack in the whileloop, the following holds*:

(1) *If* lcptab[*i*] $top_{-1}.lcp$*, 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* .

An important consequence of this Theorem is the correctness of Algorithm 2. There, the lcp-interval tree is traversed in a bottom-up fashion by a linear scan of the lcptable, while storing needed information on a stack. We stress that the lcp-interval tree is not really build: whenever an *l*-interval is processed by the generic function *process*, only its child intervals have to be known. These are determined solely from the lcp-information, i.e., there are no explicit parent-child pointers in our framework. In contrast to Algorithm 1, Algorithm 2 computes all lcp-intervals of the lcp-table *with* the child information.

Algorithm 2 (Traverse and process the lcp-interval tree).

```
1.  lastInterval := ⊥ push(<0, 0,⊥, [ ]>)
2.  for i := 1 to n do
3.      lb := i − 1
4.      while lcptab[i] < top.lcp
5.          top.rb := i − 1
6.          lastInterval := pop
7.          process(lastInterval)
8.          lb := lastInterval.lb
9.          if lcptab[i] ≤ top.lcp then
10.             top.childList := add(top.childList, lastInterval)
11.             lastInterval := ⊥
12.     if lcptab[i] > top.lcp then
13.         if lastInterval ≠ ⊥ then
14.             push(<lcptab[i], lb,⊥, [lastInterval]>)
15.             lastInterval := ⊥
16.         else push(<lcptab[i], lb,⊥, [ ]>)
```

In the next section, we will show how to solve problem merely by specifying the function *process* called in line 8 of Algorithm 2.

Bottom-up traversals

An efficient implementation of an optimal algorithm for finding maximal repeated pairs

The computation of maximal repeated pairs plays an important role in the analysis of a genome. The algorithm of Gusfield [15, p. 147] computes maximal repeated pairs of a sequence S of length n in $O(|\Sigma|n + z)$ time, where z is the number of maximal repeated pairs. This running time is optimal. In this section, we show how to implement Gusfield's algorithm using enhanced suffix arrays. This considerably reduces the space requirements, thus removing a bottle neck in the algorithm.

Let $\perp$ stand for the undefined character. We assume that it is different from all characters in Σ. Let $[i..j]$ be an l-nterval and $u = S[\text{suftab}[i]..\text{suftab}[i] + l - 1]$. Define $P[i..j]$ to be the set of positions p such that u is a prefix of S_p, i.e., $P[i..j] = \{\text{suftab}[r] \mid i \leq r \leq j\}$. We divide $P[i..j]$ into disjoint and possibly empty sets according to the characters to the left of each position: For any $a \in \Sigma \cup \{\perp\}$ define

$P[i..j](a) = \{0 \mid 0 \in P[i..j]\}$ if $a = \perp$, *and* $\{p \mid p \in P[i..j], p > 0,$ and $S[p - 1] = a\}$ otherwise.

The algorithm computes position sets in a bottom-up strategy. In terms of an lcp-interval tree, this means that the lcp-interval $[i..j]$ is processed only after all child intervals of $[i..j]$ have been processed.

Suppose $[i..j]$ is a singleton interval, i.e., $i = j$. Let $p = \text{suftab}[i]$. Then $P[i..j] = \{p\}$ and

$P[i..j](a) = \{p\}$ if $p > 0$ and $S[p - 1] = a$ or $p = 0$ and $a = \perp$, *and* $\emptyset$ otherwise.

Now suppose that $i < j$. For each $a \in \Sigma \cup \{\perp\}$, $P[i..j](a)$ is computed step by step while processing the child intervals of $[i..j]$. These are processed from left to right. Suppose that they are numbered, and that we have already processed q child intervals of $[i..j]$. By $P^q_{[i..j]}(a)$ we denote the subset of $P[i..j](a)$ obtained after processing the q th child interval of $[i..j]$. Let $[i'..j']$ be the $(q+1)$ th child interval of $[i..j]$. Due to the bottom-up strategy, $[i'..j']$ has been processed and hence the position sets $P[i'..j'](b)$ are available for any $b \in \Sigma \cup \{\perp\}$.

The interval $[i'..j']$ is processed in the following way: First, maximal repeated pairs are output by combining the position set $P^q_{[i..j]}(a)$, $a \in \Sigma \cup \{\perp\}$, with position sets $P[i'..j'](b)$, $b \in \Sigma \cup \{\perp\}$. In particular, $((p, p + l - 1), (p', p' + l - 1))$, $p < p'$, are output for all $p \in P^q_{[i..j]}(a)$ and $p' \in P[i'..j'](b)$, $a, b \in \Sigma \cup \{\perp\}$ and $a \neq b$.

It is clear that u occurs at position p and p'. Hence $((p, p + l - 1), (p', p' + l - 1))$ is a repeated pair. By construction, only those positions p and p' are combined for which the characters immediately to the left, i.e., at positions $p - 1$ and $p' - 1$ (if they exist), are different. This guarantees left-maximality of the output repeated pairs. The position sets $P^q_{[i..j]}(a)$ were inherited from child intervals of $[i..j]$ that are different from $[i'..j']$. Hence the characters immediately to the right of u at positions $p + l$ and $p' + l$ (if they exist) are different. As a consequence, the output repeated pairs are maximal. Once the maximal repeated pairs for the current child interval $[i'..j']$ have been output, the union $P^{q+1}_{[i..j]}(e) := P^q_{[i..j]}(e) \cup P_{[i'..j']}(e)$ is computed for all $e \in \Sigma \cup \{\perp\}$. That is, the position sets are inherited from $[i'..j']$ to $[i..j]$.

The algorithm runs in $O(|\Sigma|n + z)$ time. The space requirement is $O(|\Sigma|n)$ (in practice, however, the stack size is much smaller). Altogether the algorithm is optimal, since its space and time requirement is linear in the size of the input plus the output.

Computing the Ziv-Lempel decomposition

As a second application of the bottom-up traversal of the lcp-interval tree, we will describe how to compute the ZivLempel decomposition [33, 34] of a string. The Ziv–Lempel decomposition plays an important role in data compression.

For each position i of S, let l_i denote the length of the longest prefix of $S[i..n]$ that also occurs as a substring of S starting at some position $j < i$. Let s_i denote the starting position of the leftmost occurrence of this substring in S if $l_i > 0$, and $s_i = 0$, otherwise; see Fig. 3.

The Ziv–Lempel decomposition of S is the list of indices $i_1, i_2, \ldots, i_k$, defined inductively by $i_1 = 0$ and $i_{B+1} = i_{B + \max\{1,\, liB\}}$ for $B \geq 1$ and $i_B \leq n$. The substring $S[i_B..i_{B+1}-1]$, $1 \leq B \leq k$, obtained in this way is called the B th block of the Ziv–Lempel decomposition of S.

The Ziv–Lempel decomposition of a string S can also be computed *off-line* in linear time by a bottom-up traversal of the lcp-interval tree; see Algorithm 4.4. To this end, we add another value *min* of type integer to the

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	2	0	1	0	0	6	7	0
l_i	0	0	1	2	3	2	1	0	2	1	0

B	1	2	3	4	5	6	7	8
i_B	0	1	2	3	5	7	8	10
B-th block	a	c	a	aa	ca	t	at	\$

Fig. 3. The values of s_i and l_i (left) and the Ziv–Lempel decomposition (right).

quadruples stored on the stack. This value is initially set to $\perp$ and will be updated by the *process* function. At any stage, when the function *process* is applied to an l-interval $[i..j]$, all its child intervals are known and have already been processed (note that $[i..j] \neq [0..n]$ must hold). Let $[l1..r1]$, $[l2..r2]$, . . . , $[lk..rk]$ be the k child intervals of $[i..j]$, stored in its *childList*. Let $min_1, \ldots, min_k$ be the respective *min*-values of the child intervals. Let

$M = \{min_1, \ldots, min_k\} \cup \text{suftab}[q] \mid q \in [i..j]$ and $q \notin [lp..rp]$ for all $1 \leq p \leq k$.

Compute $min := \min M$ and assign for all $q \in M$ with $q \neq min$: $sq := min$ and $lq := l$. Finally, for the root $[0..n]$ of the *lcp*interval tree, we assign for all $q \in M$: $sq := 0$ and $lq := 0$.

References

[3] A. Apostolico, The myriad virtues of subword trees, in: Combinatorial Algorithms on Words, Springer-Verlag, Berlin, 1985, pp. 85–96.

[6] M. Burrows, D.J.Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital Systems Research Center, 1994.

[8] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes,
Nucl. Acids Res. 27 (1999) 23692376.

[15] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, New York, 1997.

[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, Berlin, 2001, pp. 181–192.

[33] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.

[34] J. Ziv, A. Lempel, Compression of individual sequences via variable length coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.