Linear Time Algorithms for Generalizations of the Longest Common Substring Problem

Michael Arnold* Enno Ohlebusch*

August 12, 2009

Abstract

In its simplest form, the longest common substring problem is to find a longest substring common to two or multiple strings. Using (generalized) suffix trees, this problem can be solved in linear time and space. A first generalization is the *k*-common substring problem: Given *m* strings of total length *n*, for all *k* with 2 ≤ *k* ≤ *m* simultaneously find a longest substring common to at least *k* of the strings. It is known that the *k*-common substring problem can also be solved in \( O(n) \) time. A further generalization is the *k*-common repeated substring problem: Given *m* strings \( T^{(1)}, T^{(2)}, \ldots, T^{(m)} \) of total length *n* and *m* positive integers \( x_1, \ldots, x_m \), for all *k* with 1 ≤ *k* ≤ *m* simultaneously find a longest string \( \omega \) for which there are at least *k* strings \( T^{(i_1)}, T^{(i_2)}, \ldots, T^{(i_k)} \) (1 ≤ *i_1* < *i_2* < \ldots < *i_k* ≤ *m*) such that \( \omega \) occurs at least \( x_{i_j} \) times in \( T^{(i_j)} \) for each *j* with 1 ≤ *j* ≤ *k*. In our paper, we presented the first \( O(n) \) time algorithm for the *k*-common repeated substring problem. Our solution is based on a new linear time algorithm for the *k*-common substring problem, and this note contains pseudo-code of the algorithm.

Keywords: suffix arrays, longest common substring, longest common repeat, string mining, repeat analysis

---

*Faculty of Engineering and Computer Sciences, University of Ulm, 89069 Ulm, Germany. Email: Enno.Ohlebusch@uni-ulm.de - Phone: +49 (0) 731 502 4101 - Fax: +49 (0) 731 502 4102*
Algorithm 1 Procedure lcp_update

Require: \( i \), the current position in the lcp-array
1: \( e \leftarrow LV \)
2: \( k \leftarrow 1 \)
3: while \( e.lcp \geq LCP[i] \) do
4: \( k \leftarrow k + e.size \)
5: if \( A[k].lcs \leq e.lcp \) then
6: \( A[k] \leftarrow (e.lcp, e.idx) \)
7: end if
8: last_updated \( \leftarrow e \)
9: \( e \leftarrow e.prev \) \{ previous element—the one to the left—in the doubly linked list \}
10: \( e \leftarrow e.begin \) \{ go to the first element of the interval \}
11: end while
12: create_interval(begin = last_updated, end = LV, lcp = LCP[i], size = k)
13: intptr[LCP[i]] \( \leftarrow \) last_updated

Algorithm 2 Procedure create_interval

Require: begin: first element of the interval
Require: end: last element of the interval
Require: lcp: lcp-value
Require: size: number of elements in the interval
1: begin.begin \( \leftarrow \) begin
2: begin.end \( \leftarrow \) end
3: begin.lcp \( \leftarrow \) lcp
4: begin.size \( \leftarrow \) size
5: end.begin \( \leftarrow \) begin
6: end.end \( \leftarrow \) end
Algorithm 3 Procedure list_update

Require: $i$: current position in the lcp-array
1: $\tilde{e} \leftarrow textptr[text[i]]$
2: \{\tilde{e} is the first element of the interval that contains $\tilde{e}$ and represents this interval\}
3: $\overline{e} \leftarrow intptr[LCP[RMQ(\tilde{e}.idx + 1, i)]]$
4: \{if the interval contains more than one element, not only $\tilde{e}$\}
5: if not ($\overline{e} = \tilde{e}$ and $\overline{e}.end = \tilde{e}$) then
6: \{reduce the size of the interval\}
7: $\overline{e}.size \leftarrow \overline{e}.size - 1$
8: \{if $\tilde{e}$ is the first element of the interval that contains $\tilde{e}$\}
9: if $\tilde{e} = \overline{e}$.then
10: create_interval(begin = $\overline{e}$.next, end = $\overline{e}$.end, lcp = $\overline{e}$.lcp, size = $\overline{e}$.size)
11: intptr[$\overline{e}$.lcp] $\leftarrow \overline{e}$.next
12: end if
13: \{if $\tilde{e}$ is the last element of the interval that contains $\tilde{e}$\}
14: if $\tilde{e} = \overline{e}$.end then
15: create_interval(begin = $\overline{e}$, end = $\tilde{e}$.prev, lcp = $\overline{e}$.lcp, size = $\overline{e}$.size)
16: end if
17: end if
18: \{remove $\tilde{e}$ from the list\}
19: if $\tilde{e} \neq LV$ then
20: $\tilde{e}$.prev.next $\leftarrow \tilde{e}$.next
21: $\tilde{e}$.next.prev $\leftarrow \tilde{e}$.prev
22: else
23: $LV \leftarrow \tilde{e}$.prev
24: end if
25: \{add a new element $e$ at the end of the list\}
26: $e$.prev $\leftarrow LV$
27: $e$.next $\leftarrow$ NULL
28: $LV \leftarrow e$
29: $e$.prev.next $\leftarrow e$
30: \{set the values of $e$\}
31: $e$.lcp $\leftarrow |TSA[i]|$
32: $e$.idx $\leftarrow i$
33: $e$.begin $\leftarrow e$
34: $e$.end $\leftarrow e$
35: $e$.size $\leftarrow 1$
36: $textptr[text[i]] \leftarrow e$
37: intptr[$|TSA[i]|$] $\leftarrow e$