1.2 Constructing the lcp-table in Linear Time

Next, we will present a linear time algorithm (owing to Kasai et al. [KLA+01]) that computes the lcp-table from the suffix array and its inverse. The algorithm starts with the longest suffix $S_i = S_0$ of S (so $i = 0$), computes $j := \mathsf{suftab}^{-1}[i]$ and then $\mathsf{lcptab}[j]$ by a left-to-right comparison of the characters in $S_{\mathsf{suftab}[j-1]}$ and $S_{\mathsf{suftab}[j]} = S_i$. The same is done for the other suffixes S_i of S by incrementing i successively. However, the algorithm avoids many redundant character comparisons. The idea is as follows: Consider two suffixes av and aw which directly follow each other in the suffix array. Assume that av is lexicographically smaller than aw. Suppose that the length of the longest common prefix of av and aw is ℓ. Then it is easy to see that the length of the longest common prefix of v and w is at least of length $\ell - 1$. How can we exploit this fact to compute the length of the longest common prefix of w with its direct predecessor suffix in the suffix array? Note that the direct predecessor is not always v. But we know that v is lexicographically smaller than w and all suffixes between v and w have a common prefix of length at least $\ell - 1$. Hence the longest common prefix of w with it direct predecessor is at least of length $\ell - 1$. The following Lemma precisely states this observation.

Lemma 1 $\mathsf{lcptab}[\mathsf{suftab}^{-1}[i]] \geq \mathsf{lcptab}[\mathsf{suftab}^{-1}[i-1]] - 1$.

Proof: Let h and j be the indices such that $h = \mathsf{suftab}^{-1}[i-1]$ and $j = \mathsf{suftab}^{-1}[i]$. That is, in the suffix array of S, the suffixes S_{i-1} and S_i occur at positions h and j, respectively. We have to show $\mathsf{lcptab}[j] \geq \mathsf{lcptab}[h] - 1$. First note that $\mathsf{suftab}[h] = \mathsf{suftab}[\mathsf{suftab}^{-1}[i-1]] = i - 1$. Let $k = \mathsf{suftab}[h-1]$. We proceed by case analysis. If $S_k = S_{\mathsf{suftab}[h-1]}$ and $S_{i-1} = S_{\mathsf{suftab}[h]}$ start with different characters, then $\mathsf{lcptab}[h] = 0$. Since no entry entry in table lcptab is negative, we have $\mathsf{lcptab}[j] > -1 = \mathsf{lcptab}[h] - 1$. Now suppose that S_k and S_{i-1} start with the same character, say a. Let $\omega = lcp(S_k, S_{i-1})$. Because ω is the longest common prefix of $S_{\mathsf{suftab}[h-1]}$ and $S_{\mathsf{suftab}[h]}$, we have $|\omega| = \mathsf{lcptab}[h] \geq 1$. Clearly, $\omega = a\omega'$ for some string ω'. Note that ω' is a common prefix of S_{k+1} and $S_i = S_{\mathsf{suftab}[\mathsf{suftab}^{-1}[i]]} = S_{\mathsf{suftab}[j]}$. Since the suffixes of S are lexicographically ordered in its suffix array and $S_{k+1} \prec S_i$ (this follows from $S_k = aS_{k+1} \prec aS_{i-1} = S_i$), ω' must be a common prefix of all suffixes between the indices $\mathsf{suftab}^{-1}[k+1]$ and $\mathsf{suftab}^{-1}[i] = j$ in the suffix array. In particular, ω' is a common prefix of $S_{\mathsf{suftab}[j-1]}$ and $S_{\mathsf{suftab}[j]}$. Consequently, $\mathsf{lcptab}[j] \geq |\omega'| = |\omega| - 1 = \mathsf{lcptab}[h] - 1$.

According to the preceding lemma, if $\ell = \mathsf{lcptab}[\mathsf{suftab}^{-1}[i-1]]$ is known, then one can skip $\ell - 1$ character comparisons in the computation of $\mathsf{lcptab}[\mathsf{suftab}^{-1}[i]]$. This implies the correctness of Algorithm 1.

Theorem 1 Given a string S of length n, its suffix array, and its inverse suffix array,

Algorithm 1 Computation of the lcp-table from the suffix array and its inverse.

```
ℓ := 0
for i := 0 to n − 1 do
  j := suftab^{-1}[i]
  if j > 0 then
    k := suftab[j − 1]      /* S_{suftab[k]} directly precedes S_{suftab[i]} in suftab */
    ℓ := max{ℓ − 1, 0}      /* the suffixes have a common prefix of length ≥ ℓ − 1 */
    while S[k + ℓ] = S[i + ℓ] do ℓ := ℓ + 1
    lcptab[j] := ℓ
```

Algorithm 1 constructs the lcp-table in time $O(n)$.

Proof: Algorithm 1 performs at most $2n$ character comparisons because in every every execution of the for-loop at most one redundant character comparison is made. This is because $\ell \leq n$ and the total decrease of ℓ is $\leq n$.

1.3 A Linear Time Suffix Tree Construction

Since we have now clarified how to construct the enhanced suffix array from suffix trees, we consider the other direction. We will show how the suffix tree of a string $S\$$ can be build in linear time from its suffix array and its lcp-table. In Algorithm 2, we assume that every node v in the suffix tree has a pointer $v.parent$ to its parent and a field $v.d$ that stores its distance from the root in characters. Moreover, the label of an edge (v, w) in the suffix tree is denoted by $(v, w).label$.

In Algorithm 2, $makenewleaf(j)$ creates a new leaf labeled with the leaf number j. Furthermore, $splitedge(w, v, \ell)$ takes an edge (w, v) and a natural number ℓ with $\ell < |(w, v).label|$ as input, creates a new interior node w', and splits the edge (w, v) into two edges (w, w') and (w', v), i.e., $w'.parent := w$ and $v.parent := w'$. The label $S\$[j \ldots k]$ of the edge (w, v) is also split into two labels $(w, w').label := S\$[j \ldots j+\ell]$ and $(w', v).label := S\$[j + \ell + 1 \ldots k]$. Because of the former, $w'.d := w.d + \ell$ is the distance of w' from the root.

We will show next that Algorithm 2 runs in $O(n)$ time. Let T_i denote the $\mathcal{A}^+$-tree for all suffixes $\mathsf{suftab}[0], \ldots, \mathsf{suftab}[i]$, such that the edges are ordered according to the alphabetic order. That is, after the first iteration of the for loop, T_0 consists of the root and a single leaf labeled $\mathsf{suftab}[0]$, and an edge from the root to this leaf labeled labeled $S_{\mathsf{suftab}[0]}$. Inserting the new suffix $S_{\mathsf{suftab}[i+1]}$ into T_i requires walking up the rightmost path in T_i. Each edge that is traversed ceases to be on the rightmost path in T_{i+1}, and thus is never traversed again. An edge in an intermediate tree T_i corresponds to a path