AC Automaton for the Set of Suffixes

As already mentioned, a suffix tree with suffix links is essentially an Aho–Corasick automaton for the set of all suffixes.

- We saw that it is possible to follow suffix link / failure transition from any position, not just from suffix tree nodes.
- Following such an implicit suffix link may take more than a constant time, but the total time during the scanning of a string with the automaton is linear in the length of the string. This can be shown with a similar argument as in the construction algorithm.

Thus suffix tree is asymptotically as fast to operate as the AC automaton, but needs much less space.

Matching Statistics

The matching statistics of a string $T[0..n)$ with respect to a string S is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

1. $T[i..i + \ell_i)$ is the longest prefix of T_i that is a factor of S, and
2. $S[p_i..p_i + \ell_i) = T[i..i + \ell_i)$.

Matching statistics can be computed by using the suffix tree of S as an AC-automaton and scanning T with it.

- If before reading $T[i]$ we are at the node v in the automaton, then $T[i - d..d) = S[j..j + d)$, where $j = \textit{start}(v)$ and $d = \textit{depth}(v)$. If reading $T[i]$ causes a failure transition, then $MS[i - d] = (d, j)$.

From the matching statistics, we can easily compute the longest common factor of S and T. Matching statistics are also used in some approximate string matching algorithms.

LCA Preprocessing

The *lowest common ancestor* (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v. Any tree can be preprocessed in *linear time* so that the LCA of any two nodes can be computed in *constant time*. The details are omitted here.

- Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the *longest common prefix* of T_i and T_j. Thus the lcp of any two suffixes can be computed in *constant time* using the suffix tree with LCA preprocessing.
- The longest common prefix of two suffixes S_i and T_j from two different strings S and T is called the *longest common extension*. Using the generalized suffix tree with LCA preprocessing, the longest common extension for any pair of suffixes can be computed in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms use longest common extension data structures.

Longest Palindrome

A palindrome is a string that is its own reverse. For example, `saippuakauppias` is a palindrome.

We can use the LCA preprocessed generalized suffix tree of a string T and its reverse T^R to find the *longest palindrome* in T in linear time.

- Let k_i be the length of the longest common extension of T_i and T^R_{n-i-1}, which can be computed in constant time. Then $T[i - k..i + k]$ is the longest odd length palindrome with the middle at i.
- We can find the longest odd length palindrome by computing k_i for all $i \in [0..n)$ in $\mathcal{O}(n)$ time.
- The longest even length palindrome can be found similarly in $\mathcal{O}(n)$ time.

Suffix Array

The suffix array of a text T is a lexicographically ordered array of the set $T_{[0..n]}$ of all suffixes of T. More precisely, the suffix array is an array $SA[0..n]$ of integers containing a permutation of the set $[0..n]$ such that $T_{SA[0]} < T_{SA[1]} < \cdots < T_{SA[n]}$.

A related array is the *inverse suffix array* SA^{-1} which is the inverse permutation, i.e., $SA^{-1}[SA[i]] = i$ for all $i \in [0..n]$.

As with suffix trees, it is common to add the end symbol $T[n] = \$$. It has no effect on the suffix array assuming \$ is smaller than any other symbol.

Example 4.7: The suffix array and the inverse suffix array of the text $T =$ `banana$`.

i	$SA[i]$	$T_{SA[i]}$	j	$SA^{-1}[j]$	
0	6	`$`	0	4	`banana$`
1	5	`a$`	1	3	`anana$`
2	3	`ana$`	2	6	`nana$`
3	1	`anana$`	3	2	`ana$`
4	0	`banana$`	4	5	`na$`
5	4	`na$`	5	1	`a$`
6	2	`nana$`	6	0	`$`

Suffix array is much simpler data structure than suffix tree. In particular, the type and the size of the alphabet are usually not a concern.

- The size on the suffix array is $\mathcal{O}(n)$ on any alphabet.
- We will see that the suffix array can be constructed in the same asymptotic time it takes to *sort the characters* of the text.

As with suffix trees, *exact string matching* in T can be performed by *prefix search* on the suffix array. The answer can be conveniently given as a *contiguous range* in the suffix array containing the suffixes. The range can be found using string binary search.

- If we have the additional arrays $LLCP$ and $RLCP$, the result range can be computed in $\mathcal{O}(|P| + \log n)$ time.
- Without the additional arrays, we have the same time complexity on average but the worst case time complexity is $\mathcal{O}(|P| \log n)$.
- We can then count the number of occurrences in $\mathcal{O}(1)$ time, list all occ occurrences in $\mathcal{O}(occ)$ time, or list a sample of k occurrences in $\mathcal{O}(k)$ time.

LCP Array

Efficient string binary search uses the arrays $LLCP$ and $RLCP$. For many applications, the suffix array is augmented with a different lcp array $LCP[1..n]$. For all i,

$$LCP[i] = lcp(T_{SA[i]}, T_{SA[i-1]})$$

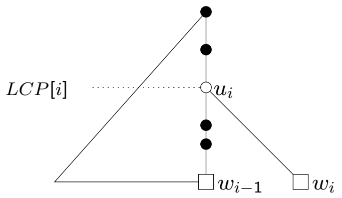
This is the same as the lcp information in the output of StringMergesort.

Example 4.8: The LCP array for $T =$ `banana$`.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$
0	6		`$`
1	5	0	`a$`
2	3	1	`ana$`
3	1	3	`anana$`
4	0	0	`banana$`
5	4	0	`na$`
6	2	2	`nana$`

The suffix tree can be easily constructed from the suffix and LCP arrays in linear time.

- Insert the suffixes into the tree in lexicographical order.
- The leaf w_i representing the suffix T_i is inserted as the rightmost leaf. The parent u_i of w_i is along the rightmost path in the tree, and the depth of u_i is $LCP[i]$. If there is no node at that depth, a new node is inserted.



- Keep the nodes on the rightmost path on a *stack* with the deepest node on top. The node u_i or the edge, where u_i is inserted, is found by removing nodes from the stack until the right depth has been reached. Note that the removed nodes are no more on the rightmost path after the insertion of w_i.

The suffix tree can be replaced by the suffix and LCP arrays in many applications. For example:

- The *longest repeating factor* is marked by the maximum value in the LCP array.
- The *number of distinct factors* can be compute by the formula

$$\frac{n(n+1)}{2} + 1 - \sum_{i=1}^{n} LCP[i]$$

  This follows from the suffix tree construction on the previous slide and the formula we saw earlier for the suffix tree.

- Matching statistics of T with respect to S can be computed in linear time using the generalized suffix array of S and T (i.e., suffix array of $S£T\$$) and its LCP array.

RMQ Preprocessing

The *range minimum query* (RMQ) asks for the smallest value in a given range in an array. Any array can be preprocessed in linear time so that RMQ for any range can be answered in constant time.

In the LCP array, RMQ can be used for computing the lcp of any two suffixes.

Lemma 4.9: The length of the *longest common prefix* of two suffixes $T_i < T_j$ is $lcp(T_i, T_j) = \min\{LCP[k] \mid k \in [SA^{-1}[i] + 1..SA^{-1}[j]]\}$.

The proof is left as an exercise.

- The RMQ preprocessing of the LCP array supports the same kind of applications as the LCA preprocessing of the suffix tree. RMQ preprocessing is much simpler than LCA preprocessing.
- The RMQ preprocessed LCP array can also replace the LLCP and RLCP arrays.

Enhanced Suffix Array

The enhanced suffix array adds two more arrays to the suffix and LCP arrays to make the data structure fully equivalent to suffix tree.

- The idea is to represent a suffix tree node v by a suffix array range corresponding to the suffixes that are in the subtree rooted at v.
- The additional arrays support navigation in the suffix tree using this representation: one array along the regular edges, the other along suffix links.

LCP Array Construction

The LCP array is easy to compute in linear time using the suffix array SA and its inverse SA^{-1}. The idea is to compute the lcp values by comparing the suffixes, but skip a prefix based on a known lower bound for the lcp value obtained using the following result.

Lemma 4.10: For any $i \in [0..n)$, $LCP[SA^{-1}[i+1]] \geq LCP[SA^{-1}[i]] - 1$

Proof. Let T_j be the lexicographic predecessor of T_i, i.e., $T_j < T_i$ and there are no other suffixes between them in the lexicographical order.

- Then $LCP[SA^{-1}[i]] = lcp(T_i, T_j) = \ell$.
- If $\ell > 0$, then for some symbol c, $T_i = cT_{i+1}$ and $T_j = cT_{j+1}$. Thus $T_{j+1} < T_{i+1}$ and $lcp(T_{i+1}, T_{j+1}) = \ell - 1$.
- Then $LCP[[SA^{-1}[i+1]] \geq lcp(T_{i+1}, T_{j+1}) = \ell - 1$.

$\square$

The algorithm computes the lcp values in the order that makes it easy to use the above lower bound.

Algorithm 4.11: LCP array construction
Input: text $T[0..n]$, suffix array $SA[0..n]$, inverse suffix array $SA^{-1}[0..n]$
Output: LCP array $LCP[1..n]$
(1) $\ell \leftarrow 0$
(2) *for* $i \leftarrow 0$ *to* $n-1$ *do*
(3) $\quad k \leftarrow SA^{-1}[i]$ $\qquad$ // $i = SA[k]$
(4) $\quad j \leftarrow SA[k-1]$
(5) $\quad$ *while* $T[i+\ell] = T[j+\ell]$ *do* $\ell \leftarrow \ell + 1$
(6) $\quad LCP[k] \leftarrow \ell$
(7) $\quad$ *if* $\ell > 0$ *then* $\ell \leftarrow \ell - 1$
(8) *return* LCP

The time complexity is $\mathcal{O}(n)$:

- Everything except the while loop on line (5) takes clearly linear time.
- Each round in the loop increments ℓ. Since ℓ is decremented at most n times on line (7) and cannot grow larger than n, the loop is executed $\mathcal{O}(n)$ times in total.