1 Concept of suffix array

Initially proposed by Mamber and Myers as an space-efficient alternative to suffix trees, suffix array has some interesting properties. In particular, it has very simple linear-time algorithm to construct directly, and suffix array is also easily converted to suffix tree.

Given example: $S = tartar$. Recall $\$ $ is alphabetically smallest. Suffix array (SA) is an array of suffix; each cell stores a suffix, and listed in lexicographic order regarding to the corresponding suffix. In this example, $SA = [ 7, 5, 2, 6, 3, 4, 1]$. One thing is straightforward. Suppose a suffix tree is given, how can we construct suffix array? We simply perform a lexicographic traversal of the suffix tree and output the suffixes (leaves) in that order. Sometimes one also wants to build suffix tree from a given suffix array. We need one additional piece of data: the LCP array (also called depth array). The depth array stores the length of the longest common string for two consecutive suffixes that are next to each other in SA. In the above example, the LCP array is $[0, 0, 2, 0, 1, 0, 3]$. Now given SA and LCP arrays, we start from smallest (first) suffix, and when move to the next, create a new branch on the tree, from the current tree. Where to branch out? This depends on the depth array. From the last added suffix $s_i$ (leaf), backup till reaching the depth of $LCP[i]$. This takes $O(1)$ per edge when backing up. Then create a new internal node if needed and then branching off an edge. Now is it possible that this branching violates the suffix tree property by creating an edge whose label has the same start symbol as another existing edge? This is not possible: it is due to the LCP property and also the lexicographic traversal. Now how about running time? The concern is that inserting a new suffix leads to traversal of sometime multiple edges. The key is that only the path leading to the rightmost suffix is traversed. Once traversed, it is no longer part of the new rightmost path, and so it will not be traversed again. All other operations are only a single action per suffix. So entire conversion takes $O(n)$.

2 Pattern matching in suffix array

The simplest way is binary search. That is, starting from the middle point $p$ of SA, we compare with $P$ against $T[p..m]$. When mismatch occurs, move $p$ to proper position just as binary search, and then repeat. This gives $O(n \log m)$ time where $m$ is the text size. But there are more advanced techniques, which gives $O(n + \log m)$ time. The main wasted time seem to be re-examining pattern from the beginning. Let $[L, R]$ denote the boundary of binary search. Then the middle point $M = (L + R)/2$ after $L$ and $R$ have been compared. Let $mlr$ to be the minimum of lengths of matches found at $L$ and $R$. Then clearly each one within $[L, R]$ also match $mlr$ and thus can start match from $mlr+1$. This alone does not ensure $O(n + \log m)$ time. To achieve $O(n + \log m)$, we need usually constant comparison per character in $P$ or once per iteration of the binary search (the number of iterations is bounded by $O(log m)$). Note: the characters in $P$ from $mlr + 1$ to the maximum of matched lengths at $L$ and $R$ have already been compared. So if we can avoid comparing these characters, we avoid repeated comparison. Definition: $LCP(i, j)$ is the length of LCP of suffixes at $i$ and $j$ of suffixes SA(i) and SA(j), where SA is the suffix array. EXAMPLE. $T = mississippi$, suffix SA(3) = issippi and suffix SA(4) = issippi. So $LCP(3, 4) = 4$. Assume all $LCP(i, j)$ have been known and accessed with $O(1)$ time. How to continue? Let $l$ and $r$ be the matched length at $L$ and $R$. 

1
The case of \( l = r \) is simple: just start at \( mlr = l + 1 \) and there is no repeat comparison. Now assume \( l > r \). Case a: if \( LCP(L, M) > l \), in this case we know the match at \( M \) is impossible since it has some longer matches with \( L \) but \( L \) can not match with \( P \). So no comparison is done and the binary search move to the next position. Case b: if \( LCP(L, M) < l \), in this case, \( L \) matches \( P \) more than at \( M \), and we do not need to compare at all (and move to left; oppose to move to right in the above case). Again, nothing is compared. Case (c): \( LCP(L, M) = l \), then we can start at \( l + 1 \) and that comparison decide where we move left or right.

Time analysis: note we only re-compare at most 1 char (case 3) in an iteration, so total will be \( O(\log m) \) such re-compare operation. The first compare of all \( P \) takes \( O(n) \).

3 How to obtain LCP values

This is Section 7.1.4.5 in Gusfield’s book. The first observation is that the number of needed LCP values is \( O(m) \). This is in fact a direct consequence of the binary search procedure. Consider the case when \( m \) is a power of 2. Then, the binary search would simply create a perfect symmetric tree, where the leaves correspond to each individual positions in \( T \) (and there is \( m \) leaves). The number of LCP needed is equal to the number of intervals (nodes) in this symmetric tree, which is clearly \( O(m) \).

Now we assume the LCP array is given. Recall that \( LCP[i] \) is equal to the length of the LCP between \( SA[i] \) and \( SA[i + 1] \). Using this, The LCP needed during the binary search can be computed based on the following observation. Let \( LCP[i, j] \) be the length of LCP of \( SA[i] \) and \( SA[j] \). Then \( LCP[i, j] = \min(LCP[i], LCP[i + 1], \ldots, LCP[j - 1]) \). You should draw a picture here to understand why this is the case. This observation allows us to have a bottom up traversal of the binary search tree; at the leaf and intervals like \([i, i + 1]\), this can be obtained directly from the LCP array. For other nodes, it takes the minimum over its two children. This clearly takes \( O(n) \) time.

4 Building the depth array in linear time

One of the remaining issue is how to efficiently build the depth array LCP. Given a suffix tree, depth array can be easily constructed (see Gusfield’s book for details). But this needs suffix tree being built before hand. This is not desirable when text is really large (like a genome). Fortunately, there is a method as follows.

Assume for now, we have a suffix array already built (that is, suffix array is built first). Consider suffix \( i \) and \( j \) are next to each other (and \( j \) is just before \( i \)) in SA. We let \( Depth(i, j) \) equal to the length of LCP between suffixes \( i \) and \( j \). Note we define \( LCP(i, j) \) in a slightly different way: the length of LCP between \( SA[i] \) and \( SA[j] \). Suppose \( Depth(i, j) > 0 \). What do we know from this assumed property? Well, \( Depth(i + 1, j + 1) = Depth(i, j) - 1 \). Next, \( Depth(i + 1, pred(i + 1)) \geq Depth(i + 1, j + 1) = Depth(i, j) - 1 \). Here, \( pred(i) \) is the suffix in SA that is immediately left to suffix \( i \). Why? Draw picture. The consequence is that we can skip the first \( Depth(i, j) - 1 \) letters, and go on to start comparison from \( Depth(i, j) \) when we compute \( Depth(i + 1, j + 1) \). Now this leads to the algorithm taking only \( O(n) \) number of comparisons. You should work out an example to make sure you understand how this algorithm works. Why? This is bounded by \( 3n \), because there are at most \( 2n \) matches, at most \( n \) mismatches (since at most one mismatch each iteration).