Lecture 8: More applications of suffix tree

1 All pairs suffix-prefix matching

Given set of strings $S_1, S_2, \ldots, S_K$ with total length $n$, find for each ordered pair $S_i$ and $S_j$ the longest suffix-prefix match of $S_i$ and $S_j$. This is one of the most frequently used algorithm for sequence assembly. We may try all pairs but when $K$ is large, the number of pairs can be very large. This will lead to $O(Kn)$ time algorithm. We now use suffix tree. First build suffix tree for all $S_i$ concatenated. Then traverse the nodes in the suffix tree and maintain a list $L(v)$ for node $v$, which holds index $i$ iff the path label to $v$ is a complete (i.e. has a empty branch to a leaf) suffix of string $S_i$. That is, $v$’s path label is a suffix of some string. How to build $L(v)$? It takes $O(n)$ to traverse the suffix tree; and the list size is bounded $O(n)$. At most $O(n)$ such complete suffixes, and each appears just in one node’s $l(v)$.

Now, for a fixed string $S_j$, traverse from root to the leaf $S_j$; then each visited node $v$, any $S_i$ s.t. $i \in L(v)$ will be a good prefix-suffix match. So we only need to maintain the deepest node depth for each string $S_i$. Time is $O(n + K^2)$, where the later term refers to the time spent in maintaining the $K^2$ entries of prefix-suffix match.

2 Finding all maximal repetitive structures

Definition: maximal repeated pair in string $S$ is pair of identical substrings and you can not extend to either side. Denoted by $(p_1, p_2, n')$ where $p_1$ and $p_2$ are start positions of the two substrings and $n'$ is the length. Example: let $S = xabcyiiizabcqabcyrxar$, where $abc$ appears 3 times. Then $(2, 10, 3)$ and $(10, 14, 3)$ but not $(2, 14, 3)$. Now we show a simple linear-time for finding all maximal repeats. As usual, build suffix tree for $S$. A simple observation: maximal repeat must appear in a node (rather than in the middle of the edge). Thus, there can be at most $n$ maximal repeats in a string of length $n$. How to find maximal repeats? Note one can not have same character to the right. How about to the left? If want to be maximal repeats, need two leaves at this node have different left adjacent character (immediately to the left). In this case, we call the node left diverse. Here, note that if a node is left diverse (LD) then all its ancestor is also left diverse. To find left-diverse nodes, a very simple approach is: each node $v$ records either it is left diverse or which same letter appear for each leaf under $v$. Then take a bottom up approach as follows. At a leaf, we just record this leaf is not LD and what is its left symbol; for an internal node $v$, if any child of $v$ is LD then $v$ is also LD; otherwise if each recorded symbol of all descendants are the same, then record it (and set $v$ to be not LD); otherwise $v$ is LD. Time is clearly $O(n)$.

Constant time lowest common ancestor. I will not explain this but it is possible to find the lowest common ancestor (LCA) of two nodes in a tree in constant time, provided that linear time preprocessing is performed on the tree. From now on, we assume such constant LCA can be found in $O(1)$ time.

3 Maximal palindrome

Given string $S$ of length $n$, find all substrings that are palindrome which are not contained by other palindromes. If one applies dynamic programming, it will take $O(n^2)$ time. We now apply the suffix tree for this problem. We assume the maximal palindrome is of even-length (the odd-length case
is similar). The key idea is building a suffix tree for $SS_r$, where $S_r$ is the reverse of $S$. Let $k$ be half-length of palindrome. The observation is, if the mid-point of palindrome is at position $q$ in $S$, we have one half copy of the palindrome at position $[q + 1, q + k]$ and in $S_r$ we have the same copy at $[n - q + 1, n - q + k]$. Recall that we can find the longest common extension (LCE) of two positions using the constant time LCA. Thus we get LCE of some length for the two position $q + 1$ in $S$ and $n - q + 1$ in $S_r$ should be exact $k$. That is, for any position $q$ (and corresponding position $n - q + 1$ in $S_r$, $k = LCE(q, n - q + 1)$. Thus, the algorithm checks each $q = 1 \ldots n$, find the LCE value $k$ as above. Each $k$ is the longest $k$ centered around $q$ (i.e. a maximal palindrome).

4 Revisiting the multiple common substring problem

Given $S_1, S_2, \ldots S_K$ (with total length $n$), we want to find the longest common substrings shared by $k$ of these $K$ strings. Recall previously we used a bitmap to manage the ancestral info but that takes $O(Kn)$ time. Now with the constant time LCA, we can now reduce it to $O(n)$ time.

We let $S(v)$ be the number of leaves under subtree $v$ (this is easy to accumulate by tree traversal), and $C(v)$ be the number of leaves from distinct strings (this is what we want!). Since $S(v) \geq C(v)$, the idea is to somehow count how many duplicates we have and then we can know $C(v)$. Definition: $n_i(v)$ is the number of leaves under subtree $v$ with identifier $i$ (from string $i$). So, the correction factor for $C(v)$ is $\sum_{i, n_i(v) > 0} (n_i(v) - 1)$. How can we find $n_i(v)$? First perform depth-first traversal and number the leaves. Here, leaves in the same subtree are consecutive. Let $L_i$ be the list of leaves with id $i$, ordered by increasing DFS number. Now the key: if we compute LCA of two consecutive leaves in $L_i$, then the LCA will not be in the subtree of $v$ if one of them is not in the subtree. Since the leaves within the subtree $v$ are consecutively labeled, there must be an interval where $n_i(v) - 1$ pairs of LCA are within the node $v$. Then we can simply just compute the LCA for all consecutive pairs of leaves and this will tell us $n_i(v) - 1$ pairs that fall within $v$. This is done by summing over for all nodes $w$ under $v$ the number of times LCA hitting $w$. However, this only deals with a single string $S_i$. We can not do for each identifier separately (which again leads to $O(Kn)$ time). Instead, we process each $L_i$ for each $S_i$; and we just accumulate (i.e. add to the number of times appear in node); and only when each $L_i$ is done, we perform depth-first traversal to count and subtract the $S(v)$ by the total sum of repetition numbers. Note that this is feasible because we only want a summation of all $n_i(v) - 1$ and so we can first get all the LCA hits and then accumulate the total number of hits below each node (coming from any $S_i$). So the total time is $O(n)$. 
