Problem 1: Isomorphism in suffix trees (25 points)

Prove the following statement:
In a suffix tree $T$, the subtree rooted at a node $u$ is isomorphic to the subtree rooted at a node $v$ if and only if there is a directed path of suffix links from node $u$ to node $v$ and the number of leaves in the two subtrees is equal.

Answer: Let the whole string is $s$ and its suffix tree $T$.

($\Rightarrow$) Suppose the subtrees at node $u$ and $v$ are isomorphic. Then they generate the same set of suffix strings in $s$, and their size is equal: $|T_u| = |T_v|$. Let one of the common strings that both subtrees generate be $w$, and $u$ is one suffix string of $s$. Let the string generated from root to $u$ be $s_u$, and the string from root to $v$ be $s_v$. We know that two strings, $s_u w$ and $s_v w$, are suffix strings of $s$. That means $s_u$ should be suffix of $s_v$, or vice versa.

Without loss of generality, let $s_v$ be suffix of $s_u$: $s_u = x_1 x_2 \ldots x_k s_v$, where $x_1, x_2, \ldots, x_k \in \Sigma$ are characters in $s$.

Let $s_i = x_i x_{i+1} \ldots x_k s_v$, $1 \leq i \leq k$. So $s_1 = s_u$, $s_k = s_v$. $s_i w$ are also suffix strings of $s$. Since $s_i$ and $s_{i+1}$ could be generated from root to some intermediate node, and they satisfy $s_i = x_i s_{i+1}$, there is one suffix link from $s_i$ to $s_{i+1}$. So there is one suffix link path which is

$$s_1(s_u) \rightarrow s_2 \rightarrow s_3 \cdots \rightarrow s_k(s_v)$$

That is one directed path of suffix link from $s_u$ to $s_v$.

($\Leftarrow$) Suppose there is one directed path of suffix links from $u$ to $v$, and the number of leaves in the two subtrees are equal, i.e., $|T_u| = |T_v|$, where $T_u$ and $T_v$ are suffix trees at node $u$ and $v$. Let the path from $u$ to $v$ be $u \rightarrow u_1 \rightarrow u_2 \cdots \rightarrow u_k \rightarrow v$.

Let the string generated from root to $u$ and $u_1$ be $w_u$ and $w_{u_1}$. Since $u \rightarrow u_1$, we know that $w_u = x w_{u_1}$. Let the suffix tree at node $u$ and $u_1$ be $T_u$ and $T_{u_1}$, and let all string set that $T_u$ and $T_{u_1}$ could generate be $W_u$ and $W_{u_1}$.

Choose any string $w \in W_u$. We know that $w_u w$ should be one suffix string of $s$. However, $w_u w = x w_{u_1} w$, and $w_{u_1} w$ should also be suffix string of $s$. This implies $w \in W_{u_1}$, because $w_{u_1}$ could only be reachable from root to $u_1$.

Since $w$ is chosen arbitrary from $W_u$, we know $W_u \subseteq W_{u_1}$. Similarly, for $u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow v$, if the string sets generated by suffix trees of $u_1 \ldots v$ are $W_{u_1}, W_{u_2} \ldots W_v$, then

$$W_u \subseteq W_{u_1} \subseteq W_{u_2} \cdots \subseteq W_v$$

However, $|T_u| = |T_v|$. That implies $W_u = W_v$, i.e., the suffix trees $T_u$ and $T_v$ are isomorphic.
Problem 2: Suffix trees and MUMs (25 points)

Given strings \(x\) and \(y\), a maximal unique match (MUM) is a substring \(w\) that occurs precisely once in both \(x\) and \(y\) and is not contained in any longer word with this property.

1. Given a threshold \(k\), describe an algorithm that produces all the MUMs in \(x, y\) longer than \(k\) using the suffix tree \(T\) built on the concatenation \(x\$1y\$2\) where \$1, \$2 are unique separator symbols.

2. Given a threshold \(k\), describe an algorithm that produces all the MUMs in \(x, y\) longer than \(k\) using only the suffix tree \(T\) built on the string \(x\$\). You are not allowed to build a suffix tree for \(y\), or add the suffixes of \(y\) to the suffix tree of \(x\).

Answer:

(1) Any substring could be recognized as prefix of suffix of the string. Let \(x = x_1 \ldots x_m\), \(y = y_1 \ldots y_n\), and let the MUM of \(x\) and \(y\) be \(x_i x_{i+1} \ldots x_j = y_p y_{p+1} \ldots y_q\), and \(x_j+1 \neq y_{q+1}, x_{i-1} \neq y_{p-1}\).

To calculate all MUMs that are longer than \(k\), first construct the suffix tree \(T\) of \(x\$1y\$2\). We see that if \(x_i \ldots x_j = y_p \ldots y_q\), such MUM should appear as prefix of two suffixes of \(x\$1y\$2\): \(x_1 \ldots x_j \ldots x_m\$1y_1 \ldots y_n\) and \(y_p \ldots y_q \ldots y_n\).

We see that the MUM should be the string generated from root to node \(u\). To ensure that MUM appears exactly once in \(x\) and \(y\), the number of \$1 should be exactly 1, and the number of \$2 should be exactly 2. Otherwise, the prefix would appear in more than one suffix, which is not MUM.

So the algorithm that produces all the MUMs longer than \(k\) is described as follows:

1. Generate the suffix tree \(T\) of the string \(x\$1y\$2\).
2. For each internal node \(u\) of \(T\), iteratively calculate the number of \$1 and \$2 in all suffix trees rooted at \(u\).
3. Enumerate all nodes of \(T\) in depth-first order.
   - If there is some node \(u\) which has exactly 2 \$2 and 1 \$1, and the string \(w\) generated from root to \(u\) satisfies \(|w| \geq k\), output \(w\) as one MUM.

(2) If only the suffix tree \(T\) of \(x\$\) is allowed, we could also search for MUM using \(T\). The idea is: for all suffix string \(y'\) of \(y\), search \(y'\) in \(T\) until stop at some node \(u\). Then the string from root to \(u\) is one possible MUM.

To ensure such possible MUM appear only once in \(x\), we need to check whether \(u\) has only one \$ in all \(u\)'s suffix trees. To check the number of occurrences in \(y\), if one such \(u\) is found, we place a mark on this \(u\). If later we encounter such marked \(u\) again, we know that the string has already be searched in another suffix of \(y\).

The algorithm could be described as follows.

1. Construct the suffix tree \(T\) of string \(x\$\).
2. For each internal node \(u\) in \(T\), iteratively calculate the number of \$ in all suffix trees rooted at \(u\).
3. Mark all nodes \(u\) in \(T\) as BLANK.
4. For all suffix string \(y' = y_k \ldots y_n\) of \(y = y_1 \ldots y_n\), \(1 \leq k \leq n\):
   - Search for \(y'\) in \(T\), until stopped at some node \(v\).
   - If \(v\) has exactly one \$ in all its suffix trees, and the string \(w\) generated from root to \(v\) satisfies \(|w| \geq k\), change the mark of \(v\) as follows.
     - If \(v\) is marked as BLANK, change it as ONCE;
If \( v \) is marked as **ONCE**, change to **DEAD**;

- If \( v \) is marked as **DEAD**, do nothing.

5. Search for all nodes \( u \) of \( T \). If \( u \) is marked as **ONCE**, output the string from root to \( u \) as one MUM.

**Problem 3: Alignment (25 points)**

An essential component in the procedure to search for a pattern \( y \) in the suffix array of \( x \) is the availability of longest common prefix information between any two suffixes of \( x \). In class, we have stated that if one has LCP for all adjacent (sorted) suffixes, he can get the LCP for any other pair of suffixes. Prove the following lemma.

**Lemma:** Let \( \text{LCP}(i, j) \) be the length of the longest common prefix of the suffixes specified in position \( i \) and \( j \) in the suffix array of \( x \). Then, when \( j > i + 1 \) we have \( \text{LCP}(i, j) = \min_{k=i,...,j-1} \text{LCP}(k, k + 1) \)

**Hint:** Show that the right-hand side of the equation above is both a lower- and an upper-bound on the left-hand side.

**Answer:**

Suffix \( i \) and \( j \) of \( x \) have a common prefix of length \( \text{LCP}(i, j) \). By the properties of lexical ordering, for every \( k \) in \( i \) and \( j \), suffix \( k \) must also have that common prefix. Therefore \( \text{LCP}(k, k + 1) \geq \text{LCP}(i, j) \) for every \( k \) between \( i \) and \( j \).

Now by transitivity, \( \text{LCP}(i, i + 1) \) must be as large as the minimum of \( \text{LCP}(i, i + 1) \) and \( \text{LCP}(i + 1, i + 2) \), extending this observation \( \text{LCP}(i, j) \geq \) as the smaller \( \text{LCP}(k, k + 1) \) for \( k \) from \( i \) to \( j - 1 \). Combined with the observation in the first paragraph, the lemma is proved.

**Problem 4: Finding maximal repeats using suffix arrays (25 points)**

A **maximal repeat** in a string \( x \) is a triple \( (i, j, l) \) such that \( x \) contains a repeat of length \( l \) starting at positions \( i \) and \( j \), and this repeat cannot be extended further to the left or right. Formally, \( x[i:i+l-1] = x[j:j+l-1], \) but \( x[i-1] \neq x[j-1] \) and \( x[i+l] \neq x[j+l] \). Given a string \( x \), design an algorithm that finds the longest maximal repeat in \( x \) in time \( O(|x|) \), using a suffix array.

**Answer:**

First use a linear-time construction to build the suffix array of \( x \), which also computes the LCP array.

Given a maximal repeat \( (i, j, l) \), suffix \( i \) and suffix \( j \) will be adjacent in the suffix array, with an LCP of \( l \). Thus, it is sufficient to scan the LCP of adjacent suffixes, and find the largest entry. The time complexity is linear in the size of \( x \).