COMP 251: Data Structures and Algorithms

Solution for Assignment #3

1 Suffix Tree

1.1 Longest Repeated Substring

By using the Build-Suffix-Tree routine, we assume that we are given the suffix tree T of the input string. First, make the following observation: If there are two $root - leaf$ paths p and q, such that they share any edges(vertices), the edges(vertices) themselves form a $root - leaf$ path. (This follows from the fact that trees are acyclic.) Moreover, since both p and q are substrings of the input string, the new path formed by $p \cap q$ corresponds to a repeated substring. With this observation in mind, we merely need to look for the deepest branching node in the tree T, where the depth of a node v is measured by the number of characters from root to v.

This can be easily done via Depth-First-Search. First, start by marking the root node with zero. Initialize a varaible *max_distance* to zero, and *candidate* to the root vertex. Now, run DFS on the tree. As the algorithm recursively visits the vertices, it computes the distance from root to the newly visited vertex v by the equation:

$$distance(v) = distance(v.parent) + length(v)$$

where $length(v)$ indicates the length of the string contained in v. Furthermore, if this vertex is a branching node(that is, it has more than or equal to two children nodes), check if its distance is greater than current *max_distance*. If so, update *max_distance* and set $candidate = v$.

Once you compute this DFS, the value stored in *max_distance* is the length of longest repeated substring, and *candidate* contains the vertex such that the characters along the root to *candidate* vertex correspond to the longest repeated substring. Simply output this string. You might have to run another DFS to find the path from root to candidate to output the string - that's OK, since we are not hurting the asymptotic running time. On the other hand, you must be careful not to "store" the entire candidiate substring as you're running the DFS. The length of the candidate substring can be $O(length[A])$, so updating this string each time may throw in another linear amount of work within your DFS, causing the running time to blow up to $O(n^2)$.

1.2 Longest Common Substring

In this problem, the same suffix tree constructed from the Build-Suffix-Tree routine can be used without modification. The trick is to use another terminal character. First, construct a new string A by concatenating A_1 and A_2 as follows:

$$A = A_1\#A_2\$$$

Then, call the Build-Suffix-Tree(A) to construct the suffix tree for A. Now, let's look at what we have in the suffix tree. As before, if there are two root-to-leaf paths p an q, the path $p \cap q$ is a common substring of the two strings along p and q. It is not, however, that all such common substrings are common substrings of A_1 and A_2: they might be from repeated substring from a single string. Now, we observe the following: Let p and q be two root-to-leaf paths that share edges along $p \cap q$. Then, we have that $p - (p \cap q)$ contains \$, and $q - (p \cap q)$ contains # (but not \$), if and only if $p \cap q$ is a common substring of A_1 and A_2. This condition simply forces the substring $p \cap q$ to be in both A_1 and A_2.

Thus, we now need to look for the deepest branching vertex in T, such that one of its child contains \$, and one of its child contains #(without \$). The rest of the algorithm stays the same with the previous question, except that here we need to recursively check if 1) there is a child that contain \$, and 2) there is a child that contains #, but not \$.

You can do this by a single DFS, which runs linear in the size of the tree. The size of the tree constructed by Build-Suffix-Tree routine is $O(length[A_1] + length[A_2])$, and thus the running time as desired.