CAP 6515 ASSIGNMENT #2

1 Solution to Question No: 1

1.1 Question I

Ukkonen's algorithm constructs the suffix tree in **n** phases for a given string **S** of length **n**. Initially, it builds T_1 by adding root node and one leaf node at one edge which is labeled with the first character of string. Then, at each phase **i+1**, it builds a new implicit suffix tree T_{i+1} based on the tree T_i. T_i is the implicit suffix tree for **S[1...i]**. The pseudocode of Ukkonen's algorithm which constructs the suffix tree for string **S** in linear time (O(n)) is shown in Algorithm 1.

Time Complexity Analysis: The outer loop runs m-1 times where m is the length of the string. For (i+1)th phase, there are (i+1) extensions. But for the extensions following Rule1, only a global value has to be incremented which takes only O(1) time. If an extension is following Rule3 then nothing has to be done. The algorithm only need to add leaf nodes(internal nodes might or might not be added) when the extensions are following Rule2. That means every time any extension follows Rule2, a leaf node is create. In the final implicit suffix tree, number of leaf nodes $<=$ length of string m. That means, throughout the algorithm, Rule2 is followed not more than m times which makes this algorithm a linear time (O(m)) algorithm.

1.2 Question II

The implicit suffix tree for string "xabxababxba" has been show step by step (for each phase i+1) in the following pages. Also, the list of rules used for each phase (i+1) and each extension (j) to construct the suffix tree (using the Ukkonen's algorithm) has been shown.

Algorithm 1 Ukkonen's Algorithm

```
m ← length(S)                                                  ▷ Length of Text String S
Construct tree T_1                          ▷ Contains root, one leaf node, one edge; label of edge is the 1st character of S
No_of_Leaf = 1                                                 ▷ Global variable that keeps track of number of leaves
Last_Edge_Pos = 1                                              ▷ Global variable that keeps track of last character of leaf edges
ActiveNode = root
ActiveEdge = S[1]
ActiveLength = 0                                               ▷ Current active point (root, a, 0)
Node_v = NULL                                                  ▷ Used to create suffix link

for i ← 1 to m-1 do                                            ▷ Construct T_{i+1} from T_i
    Last_Edge_Pos++                                            ▷ Apply Rule 1; increment global variable by 1 for all leaves

    for j ← No_of_Leaf+1 to i+1 do                             ▷ Begin extension j for phase i+1
        if ActiveLength = 0 then
            ActiveEdge = S[i+1]
        Search for path P from ActiveNode that starts with ActiveEdge   ▷ using Hashing technique takes O(1) time
        if Path P Exists then
            E = P
        else
            Create new leaf and label leaf edge with S[i+1]                                  ▷ Rule 2
            No_of_Leaf++
            Continue

        while ActiveLength ! < length(E) do   ▷ ActiveNode will change during walk down, ActiveEdge
                                                  and ActiveLength will change accordingly
            ActiveNode = ActiveNode− >Next                                                   ▷ Skip to next node
            ActiveEdge = S[Postion of ActiveEdge + length(E)]
            ActiveLength = ActiveLength - length(E)
            E = edge that comes out of ActiveNode and starts with ActiveEdge character

        ActivePostion = Postion of first character of edge E+ ActiveLength
        C= S[ActivePostion]

        if C = S[i+1] then                  ▷ Rule 3: S[j..i+1] correspond to a path, no walk down required
            ActiveLength++
            Break                                                                            ▷ Moves to next phase
        else                                                                                 ▷ Rule 2
            Create Internal node V before ActivePostion                                      ▷ Divides edge E
            Create new leaf node and connect it to V with an edge labelled with S[i+1]
            No_of_Leaf++
            if Node_v ! = NULL then   ▷ Create Suffix link from previous to current internal node
                Create suffix link pointing from Node_v to new internal V
            Node_v = V                ▷ Suffix link of currently created internal node will be set in next extension

            if ActiveNode = Root then
                ActiveLength−−
                ActiveEdge = S[j+1]
            else                                                                             ▷ ActiveNode not root
                ActiveNode = New node pointed by suffix link                                 ▷ Follow suffix link from ActiveNode
    if No suffix link created from Node_v  then
        Create suffix link pointing from Node_v to root
    Node_v = NULL
return T_m                                                     ▷ Returns Final Suffix Tree
```

2 Solution to Question No: 2

2.1 Problem Definition:

In Ukkonen's algorithm, the run time is O(n) if the alphabet size ($\sum$) is constant. In case of constant alphabet size, the search for which branch to select next from any node can be done in O(1) time using Hashing technique. If we search for the branch naively, it will take O(n) time which will make Ukkonen's algorithm's total time complexity O(n^2). But to apply Hashing we need to do the mapping and that requires a constant alphabet size. If alphabet size is variable, then we cannot perform Hashing which will increase the time complexity of Ukkonen's algorithm to O(n^2).

2.2 Solution:

For searching the path, if we use Binary Search instead of Linear Search, that will decrease the complexity. Complexity of Binary Search will be O(log$\sum$) where $\sum$ is the alphabet size. If the alphabet size is comparable to the length of the input string n, then complexity of Binary Search will be O(logn). In line no 16 of the Algorithm 1 (given in Question 1(I)), we have to use Binary Search instead of Hashing. For that, while creating a new node from any internal node or root, we have to maintain an order (lexicographic for English alphabets). Thus, the total complexity of Ukkonen will be O(nlogn) for a variable alphabet size (comparable to string length).

3 Solution to Question No: 3

3.1 Problem Statement:

A simple version of the peptide vaccine design problem has to be formulated as the shortest unique substring problem, which attempts to nd the shortest peptide in the proteins of the pathogen that are not a part of any protein from the host (human).

3.2 Solution:

Suppose, the sequence of protein in pathogen is P (length m) and the sequence of protein in Human is S (length n). The steps to find out the shortest unique substring between H and P are given below:

- First, we have to add a special symbol ($) at the end of the sequence H.
- Then we have to build a suffix tree for H$. This can be done in linear time (O(n)) using the Ukkonen's algorithm.
- Next we have to insert the possible suffixes of sequence P in the existing suffix tree using hashing which will generate a larger suffix tree with more nodes. As we are using hashing, this will also take linear time (O(m)).
- After that, from the suffix tree we have to start walking down from the root until we reach the edge that doesn't contain any special symbol ($) in it. There can be multiple such edges, means there can be multiple such paths from the root.If the length of string is N, then the maximum possible number of nodes in an implicit suffix tree is N+1 (in worst case). So, in worst case, to look through all the possible paths starting from the root, the algorithm will need to visit N+1 nodes. So the searching time won't me more than linear. This will return us all the possible unique suffixes of P. From that we can chose any suffix with minimum length. This will return us all the possible solutions.

  However, this can be done more efficiently if we need only one solution. We have to consider the newly created leaf nodes only (ones that didn't exist in the implicit suffix tree of H$). Now, we can sort the new leaf nodes according to their depth (sorting time <= O(mlogm)). If we need only one solution (multiple shortest unique substring may exist), then only visiting the path of the leaf with minimum depth up to root will gives us our solution. In this way the number of nodes visited for the search operation will be <<N, again searching time will be linear. From the new leaf with minimum depth, we have to go to its immediate parent node. Each node will store 2 index values, starting and ending index of the characters represented by it's edge (in edge). From that we will know the index positions of the characters that are present in the shortest unique substring. We only have to consider the index positions (start, end) of the internal nodes up to root. Only exception is the immediate parent node of the leaf, for that we have to consider index positions from start to end+1.
- After reaching such an edge (doesn't containt $), we just have to add the first character of that edge to the path that we have visited so far. At this, point we might have multiple paths with multiple lengths.
- Among the selected paths, the paths which have the minimum length will be the shortest unique substring between P and H. There can be multiple possible solutions.

3.3 Example:

Suppose, H = ACDBCA, H$ = ACDBCA$, P = ACBDA
Suffix Tree S for H$ and updated Suffix Tree after adding the suffixes of P is given below in Figure 1 and 2 respectively:
Edges without the symbol $:
BDA, BDA, DA, A
Paths starting from Root and adding first character from the edges:
ACB, CB, BD, DA
Shortest unique substring:
CB, BD, DA

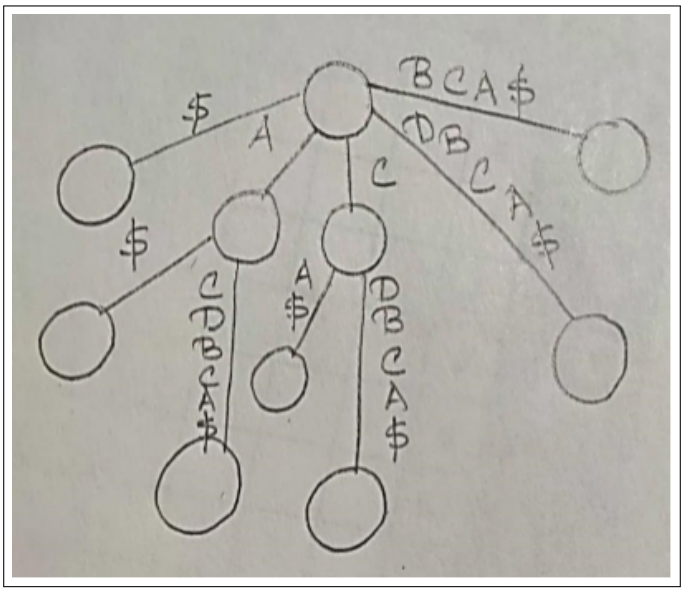

Figure 1: Suffix Tree S for H$

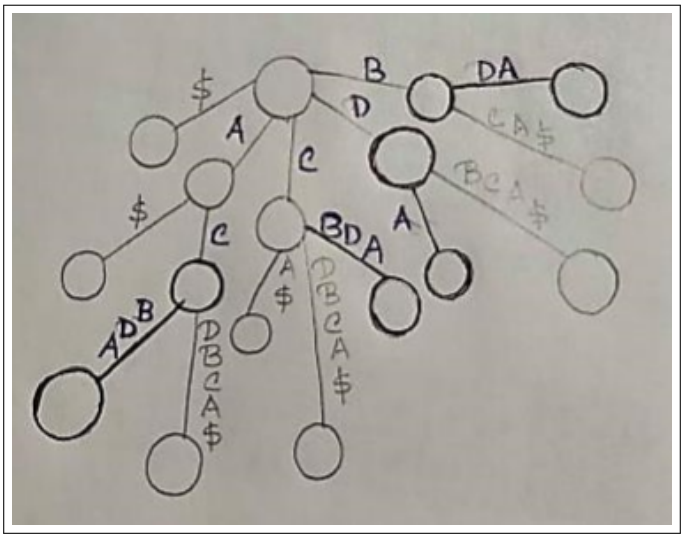

Figure 2: Suffix Tree S after adding suffixes of P. Here Blue characters represent suffixes of P and black nodes and edges were added for P

3.4 Complexity:

The time required to build the suffix tree for H is O(n). Then another O(m) time will be required to add suffixes of P in that tree. So, total O(m+n) time to build the tree. Then walking down the tree starting from root for each suffix of P will also take linear time (O(n)) as maximum depth of the tree is n (or m if $m > n$). So, considering $n > m$, approximate time complexity is O(n+m+n) or O(3n) which is equivalent to O(n). So this can be done in linear time.

4 Acknowledgment

To understand suffix tree and how Ukkonen's algorithm builds suffix tree in linear time, I read Chapter 6.1 from 'Algorithms on Strings, Trees, and Sequences' by Dan Gusfield and also the article on Suffix Tree from GeeksforGeeks.