Indian Institute of Technology Ropar
भारतीय प्रौद्योगिकी संस्थान रोपड़

Department of Computer Science Engineering
Data Structures and Algorithm

Bachelor Project

Ukkonen Suffix Tree Implementation and Application

Devanshu Verma(2021CSB1083)
Divyankar Shah(2021CSB1086)
Doodh Nath Tiwari(2021CSB1087)

Time frame: oct 2022 - nov 2022

Supervisor
Prof. Dr. Anil Shukla

Advisor
Sravanthi Chede

1 Introduction

Suffix Tree is a data structure that is very useful for text processing.It is a compressed trie containing all the suffixes of a text.It allows for fast implementation of many text operations. A suffix tree of n characters is a rooted tree that contains exactly N leaves provided that last character in string is unique in string.

Using UKKONEN algorithm, it can be implemented in O(n) time.This also allows for application of tree for many uses such as finding common substring etc. in linear time.However the space required is large to store suffixes of a text.

Suffix tree for sting of lenghth n is defined by:

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.
- Each edge is labelled with a non-empty substring of S.
- Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i. . . n].

Since such a tree does not exist for all strings we append a special terminal symbol not found in string(generally it is $). Suffix links are a key feature.All internal nodes in tree have a suffix link pointing to another node.If the path from the root to a node spells the string XA, where X is a single character and A is a string (possibly empty), it has a suffix link to the internal node representing A.

A generalized suffix tree is a suffix tree made for a set of strings instead of a single string. It represents all suffixes from this set of strings. Each string must be terminated by a different termination symbol.

2 Implementation

The algorithm begins with an implicit suffix tree containing the first character of the string. Then it steps through the string, adding successive characters until the tree is complete.

UKKONEN's algorithm constructs an implicit suffix tree Ti for each prefix S[1...i] of string S. It first builds T1 using first character, then T2 using second character, then T3 using third character, ..., Tn using the nth character.

You can find the following characteristics in a suffix tree that uses Ukkonen's algorithm:

- Implicit suffix tree Ti+1 is built on top of implicit suffix tree Ti.
- UKKONEN's algorithm is divided into n phases(one phase for each character in the string).
- Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1...i+1].

2.1 Extension Rules

There are three extension rules:

1. If the path from the root labelled S[j...i] ends at a leaf edge(meaning S[i] is last character on leaf edge), then character S[i+1] is just added to the end of the label on that leaf edge.
2. If the path from the root labelled S[j...i] ends at a non-leaf edge (means there are more characters after S[i] on path) and next character is not S[i+1],then a new leaf edge with label S[i+1] and number j is created starting from character S[i+1].A new internal node will also be created if S[1...i] ends inside a non-leaf edge.
3. If the path from the root labelled S[j..i] ends at a non-leaf edge and the next character is S[i+1],do nothing.

One important point to note is that from an internal node,there will be one and only one edge starting from one character.

Following is a step by step suffix tree construction of string xabxac using UKKONEN's algorithm:

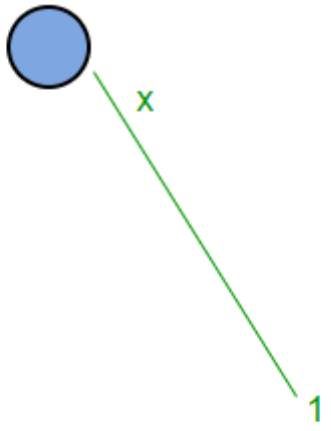


Figure 9 : T1 for S[1...1]
Adding suffixes of x(x)

Rule2-A new leaf node

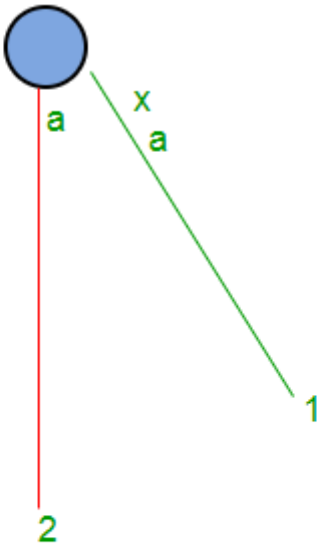


Figure 10 : T2 for S[1...2]
Adding suffixes of xa(xa and a)

Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

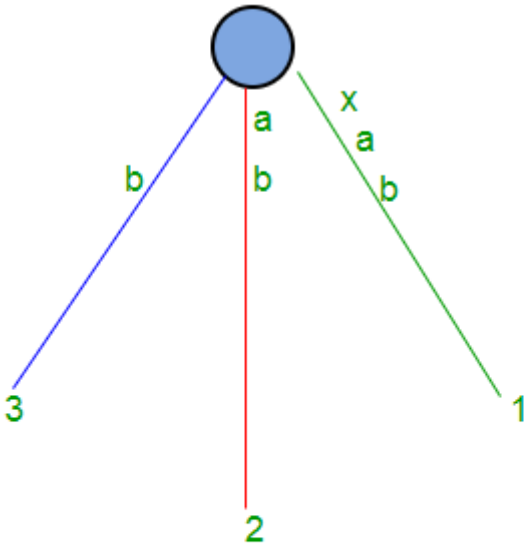


Figure 11 : T3 for S[1...3]
Adding suffixes of xab(xab,ab and b)

Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

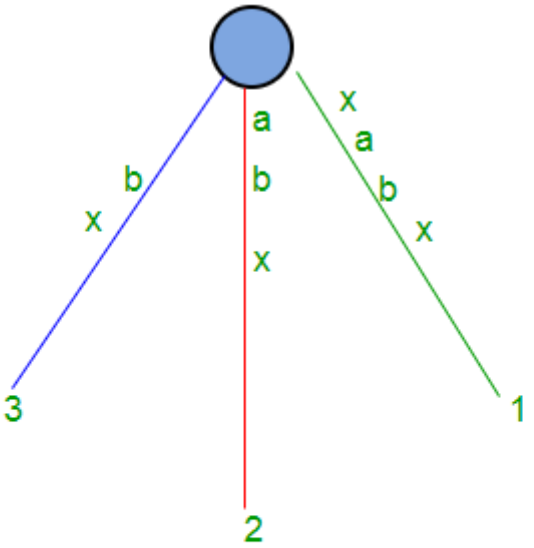


Figure 12 : T4 for S[1...4]
Adding suffixes of xabx(xabx,abx,bx and x)

Rule1-Extending path label in existing leaf edge
Rule3-Do nothing(path with label x already present)

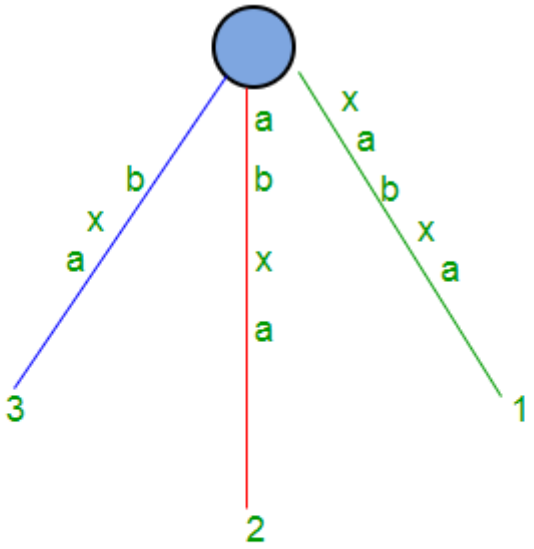


Figure 13 : T5 for S[1...5]
Adding suffixes of xabxa(xabxa,abxa,bxa,xa and x)
Rule1-Extending path label in existing leaf edge
Rule3-Do nothing(path with label xa and a already present)

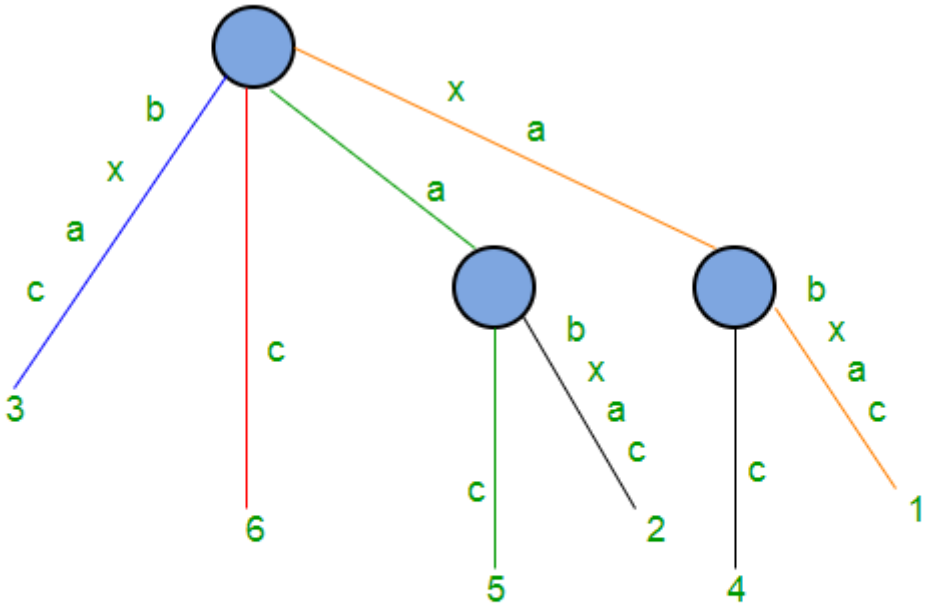


Figure 14 : T6 for S[1...6]
Adding suffixes of xabxac(xabxac, abxac, bxac, xac, ac, c)
Rule1-Extending path label in Existing leaf edge.
Rule2-Three new leaf edges and two new internal nodes

In each phase i we add ith character to the tree created so far(i.e. $T_(i-1)$),total extensions are done in each phase i. To do the jth extension of phase i+1,we first need to find end of path S[j...i] in current tree.One way is start from root in every extension and traverse the path matching S[j..i].This will take O(n^3) time to build the suffix tree. Suffixlinks come into action to solve above problem.

2.2 Suffixlink

For an internal node v with path-label xA,if there is another node s(v) with path-label A,then a pointer from v to s(v) is called a suffix link. Here x is a single character and A is a substring(possibly empty). If A is empty substringstring, suffix link from internal node will go to the root node.

In extension j of some phase i,if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labelled A already ends at an internal node.
- OR a new internal node at the end of A will be created.

In extension j+1 we will create suffix link from internal node created in jth phase to new internal node of extension j+1.

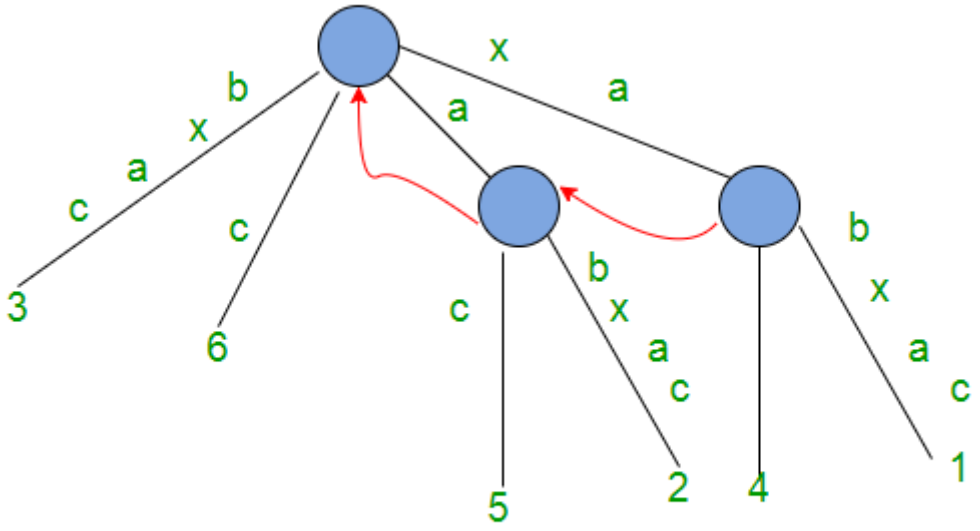


Figure 15 : Suffix links in red arrows

So in extension j of phase i+1,we start from end of path s[j-1...i], walk up one node and follow suffix link and then walkdown.

2.3 Skip/Count trick

When walking down from node s(v) to leaf,we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel,skip to the last character on that edge.

Using suffix link along with skip/count trick, suffix tree can be built in O(n^2) as there are n phases and each phase takes O(n) time.

2.4 Edge label compression

To restrict space complexity to O(n) instead of storing actual characters we store start and end indexes on each edge that denote the starting and end position of pathlabel.

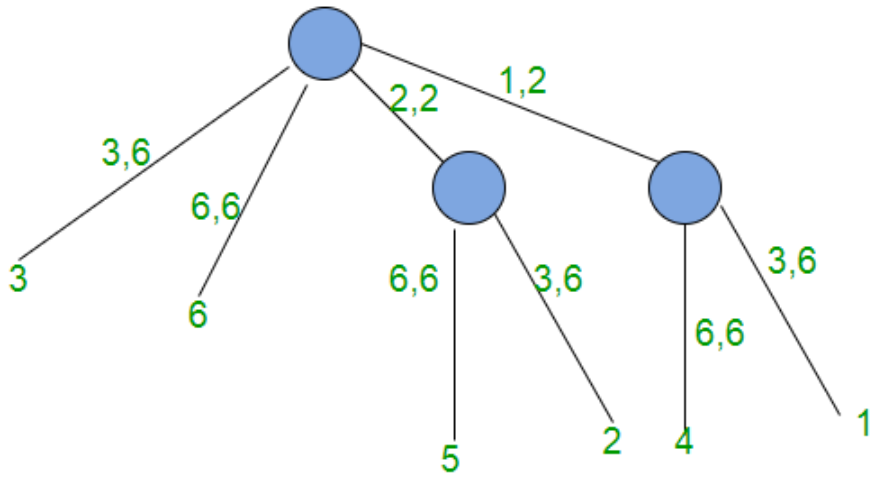


Figure 19 : Suffix tree for string xabxac with edge-label compression
Figure 14 shows same suffix tree without edge-label compression

2.5 Trick 2

once extension rule 3 is applicable in an extension j of phase i then it will also be applicable in further extensions of the same phase.That's because if path labelled S[j..i] continues with character S[i+1], then path labelled S[j+1..i], S[j+2..i], S[j+3..i],. . . , S[i..i] will also continue with character S[i+1].

So no more work need to be done in current phase as soon as extension rule3 applies.

2.6 Trick3

Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions.Once a leaf is labelled as j,extension rule 1 will always apply to extension j in all successive phases.

So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i, extension $J_{i_{+1}}$ onwards, rule 2 applies to zero or more extensions and then finally rule 3, which ends the phase.

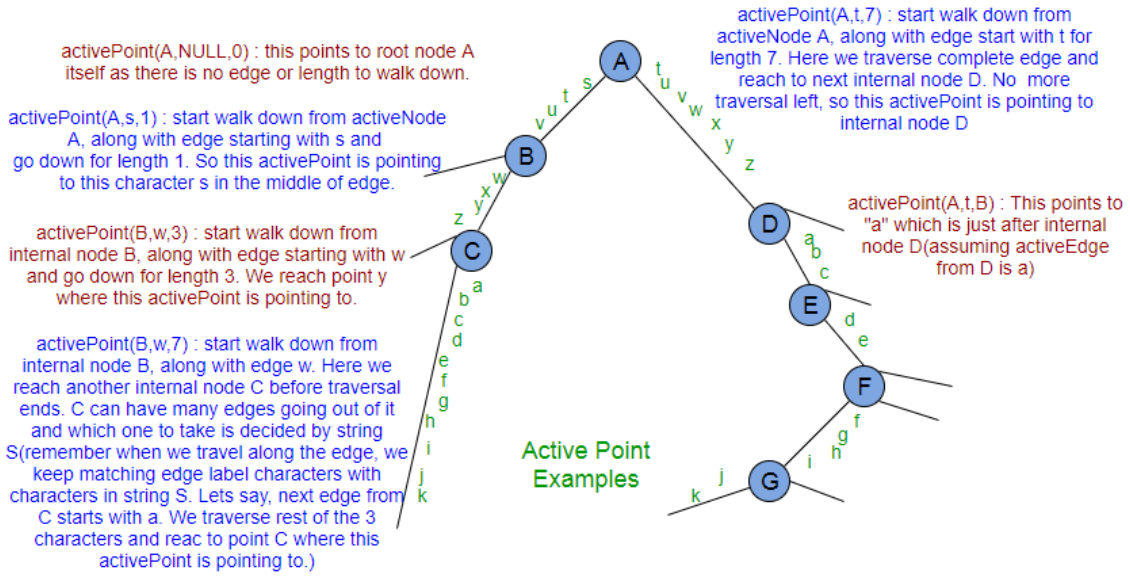
2.7 Trick 4

To implement what we discussed in trick 3 about extension rule 1.Since in phase i, all leaves have end index i, so we maintain a global variable leafend, and in each successive phase increment that index by 1.This takes constant O(1) time.

2.8 activePoint

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied. activepoint contains three variables: activeNode, activeEdge and activeLength.

- activeNode: This could be root node or an internal node.
- activeEdge: When we are on a node and need to walk down, we need to know which edge to choose.activeEdge stores that. In case that activeNode itself is the point from where traversal starts,then activeEdge will be set to next character being processed in next phase.
- activeLength: This tells how many characters we need to walk down on the path represented by activeEdge from activeNode to reach the activePoint where traversal starts.In case, activeNode itself is the point from where traversal starts,then activeLength will be 0.



2.9 activePoint changes

1. activePoint change for extension rule 3(APCFER3): When rule 3 applies in any phase i, then before we move on to next phase i+1, we increment activeLength by 1.There is no change in activeNode and activeEdge.

2. activePoint change for walk down (APCFWD):activePoint may also change when we do walk down.Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLenght as appropriate so that new activePoint represents the same point as earlier).

3. activePoint change for Active Length ZERO (APCFALZ): At start of an extension, when activeLength is ZERO, activeEdge is set to current character being processed, because there is no walk down needed here and so the next character we look for is current character being processed.

4. activePoint change for extension rule 2 (APCFER2): Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrease the activeLength by 1 and activeEdge will be set "S[i – remainingSuffixCount + 1]" where i is current phase number.activeLength is decreased by 1 because activePoint gets closer to root by length 1 after every extension. Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge.

3 Applications

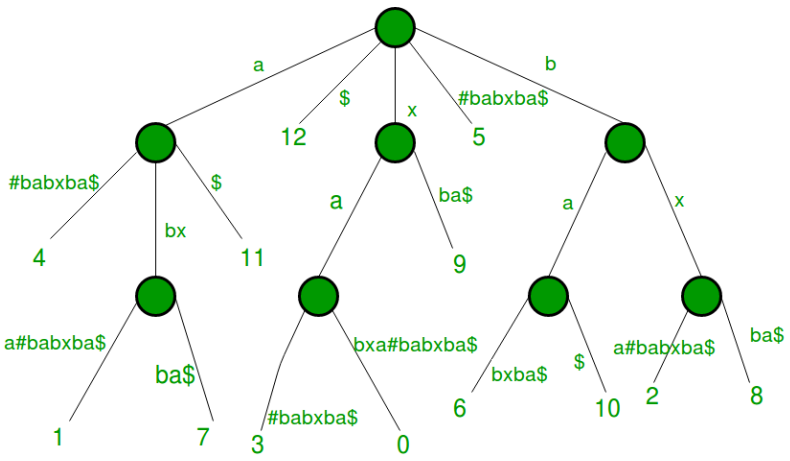
1.) Substring check : Suffix tree can be used to search for a pattern inside given text in O(m) time where m is the string length if the pattern that we are searching for. Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

2.)Longest repeating substring : In a suffix tree, one node cannott have more than one edge starting with same character , so , if there are repeated substring in the text, they will share same path and that path in suffix tree will go through one or more internal node(s) down the tree.

longest repeated substring will end at deepest internal node.So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

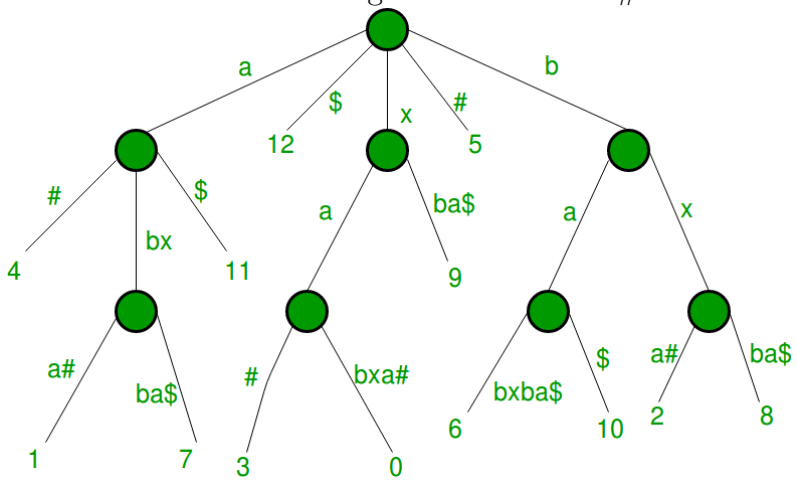
3.)Longest common substring : We will first build generalised suffix tree for two strings by appending string1 and string2 as "string1#string2$" and creating suffix tree. If a path label has "#" character in it, then we are trimming all characters after the "#" in that path label.

consider below example:



Suffix tree for string xabxa#babxa$

after removing characters after #:

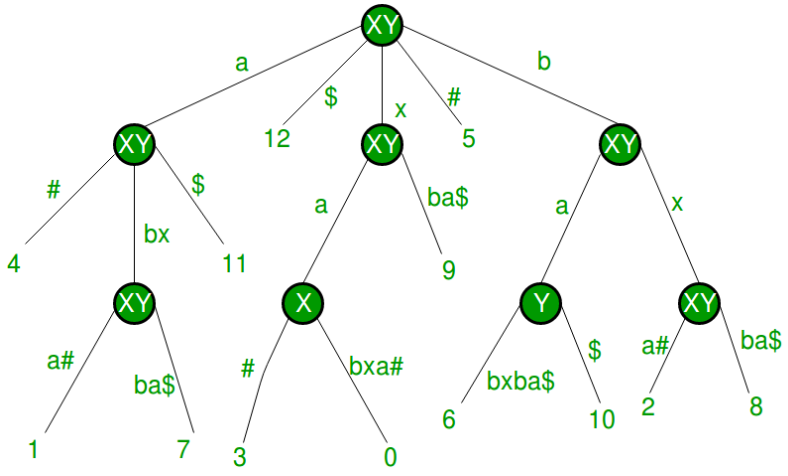


This is generalized suffix tree for xabxa#babxba$ In above, leaves with suffix indices in [0,4] are suffixes of string xabxa and leaves with suffix indices in [6,11] are suffixes of string babxa.

In suffix tree above there are internal nodes having leaves below it from:

- both strings X and Y
- string X only
- string Y only

Nodes marked XY are for substring common in both strings.We run dfs over all nodes and find deepest internal node marked XY, which gives us the solution



4.)Longest Palindrome substring: This application is similar to LCS application discussed above. To implement this application we take string1 as original string and string2 as reverse of string1 and then create generalised suffix tree of them.Now we find LCS between them such that LCS in string1 and string2 is from same position in string1.For this we create unordered set that store starting indexes of original and reverse string and check if node satisfied condition necessary.

If there is a common substring of length L at indices Si (forward index) and Ri (reverse index) in string1 and string2,then these will come from same position in S if N–Si-L–1=Ri where N is string length.

4 Time and space complexity

The algorithm consists of n phases, where n is the length of the string. In phase i, the algorithm processes the i-th character of the string.

Each phase i has i extensions, corresponding to the suffixes ending at the current character. The total number of extensions across all phases is given by the sum 1+2+3+...+n, which is n(n+1)/2.

Suffix links allow the algorithm to move efficiently between different parts of the tree, avoiding redundant work. Due to the use of suffix links, the work done in each extension is amortized to O(1).

Even though the naive approach might suggest $O(n^2)$ complexity due to the sum of extensions, suffix links reduce this to O(n), as each character is processed in constant time.

Following table shows the time complexities of implementation and applications of suffix tree:

5 Conclusion

By implementing Suffix tree we can see how useful this data structure is for text processing considering that it can solve various complex problems such as those dicussed above, in linear time which is super effecient compared to many other data structures used for same.Also as we saw that it also takes low space in addition to fast computation.

Operation	**Time Complexity**	**Brief explaination**
Construction	O(N)	N is the number of characters in text.
Substring check	O(m)	m is the number of characters in pattern to check and since comparison happens for each character of pattern one by one,so overall complexity is O(m)
Longest Repeated Substring	O(N)	We are basically running dfs on nodes of tree,so each node is visited atmost once,hence worst case complexity is O(N), N denotes the number of characters in text.
Longest common substring	O(M+N)	Again we are running dfs search on nodes of tree of appended string of length M+N,where M and N are lengths of first and second strings respectively.
Longest Palindrome Substring	O(N)	Since we are basically finding LCS between a string and its reverse string so,M=N so O(2N) and for large numbers we ignore 2 and say O(N)

Table 1: Time complexity table

6 References

1. https://rosettacode.org/wiki/Ukkonen's_suffix_tree_construction
2. https://en.wikipedia.org/wiki/Suffix_tree
3. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/?ref=rp
4. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-2/
5. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/
6. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-4/
7. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/
8. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/
9. https://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/
10. https://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/
11. https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/
12. https://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/
13. https://favtutor.com/blogs/ukkonen-algorithm-suffix-tree