Suffix Tree

MTL342

Vivek Muskan

2017MT10755

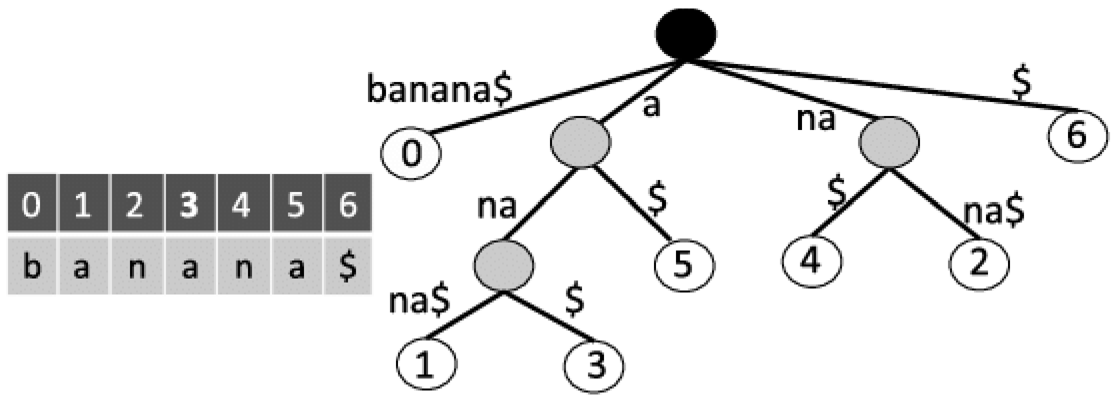
Sem I 2019-20

Table of Content

1. Introduction

Definition : *A **Suffix Tree** for any word 'w' is defined as a tree in which every path from root to leaf is a suffix of word 'w'.*

Suffix tree is the Trie of all the possible of suffix of any word.
i.e. Consider a string X: "banana$"
So, the suffix tree of X is a TRIE of {"banana$", "anana$", "nana$", "ana$", "na$", "a$", "$"}



Suffix tree have some basic properties which increases the efficiency of this data structure and helpful in a lot application like pattern searching, common substring problems etc.

Definition: *A suffix tree is called an **implicit suffix tree** if any one of its suffixes is a prefix of another suffix.*

Definition: *A suffix tree is called an **explicit suffix tree** if none of its suffix is a prefix of another suffix.*
TRIE of all suffixes of a word requires some modifications to make it a suffix tree.

Modification in TRIE of suffixes

a) We add a special character '$' at the end to convert an implicit suffix tree into explicit suffix tree.
b) We merge all the nodes of the tree which has a single child to reduce height.

After these modifications we will get a data structure shown in fig1.

2. Naïve Implementation

A simple naïve implementation algorithm will be that we generate all the possible suffixes and then add them all in a TRIE.

Algorithm:

```
buildSuffixOf(S[1….m])
    For i = 1 to m :
        Current_suffix = S[i,m]
        addToTrie(Current_suffix)
    end
end
```

Now it adds the string *Current_suffix* O(m) times. Let us now look at *addToTrie()*

```
addToTrie(S[1….m])
    int j = 1
    while (Trie.contains(S[j]))
        keepMatching()
    end
    For i = j to m
        makeNewNode(S[i])
    end
end
```

Time Complexity

Clearly each time we add a string we have to traverse through the trie until we get to the end of Trie and then add the remaining characters of string, which takes O(m) time and since it add O(m) of such strings,

So, overall Time complexity of this method is O(m^2).

Space Complexity

Since total m suffixes are possible of a string of length m hence a maximum of O(m) path are possible in the suffix tree from root to leaf.

But size of each node is not constant but is a function of m, because each node contains a string whose length are of O(m).

Hence total space complexity of this method is O(m^2).

3. McCreight's Algorithm

Idea

Let us call tree T_i be the suffix tree of suffixes S[1,m], S[2,m],… S[i,m]. The idea is to generate T_{i+1} from T_i. High level version of extension function of McCreight's Algorithm is given below

Algorithm

```
generateNextof ($T_i$)
    int j = i+1
    while($T_i$.contains(S[j])
            keepMatching()
    else
            splitNewLeaf(S[j,..m])
    end
end
```

Time Complexity

The functions used in this algorithm like

contains() : can have O(m) because T_i have O(m) number of children

keepMatching() : can have O(m) because suffixes of S have O(m) length

splitNewLeaf() : It is O(1) time because in each generation we have to add only one new suffix.

Overall Time complexity could be O(m^2) but McCreight's algorithm uses the concept of suffix link and compact suffix tree to reduce the time complexity to O(m) and even space complexity to O(m). Let us see how it can be done.

Suffix Link

Definition : *If a node v contains string xA and another node u contains string A then we add a link from node u to v and we call it **Suffix Link**, where A can be any non-empty string.*

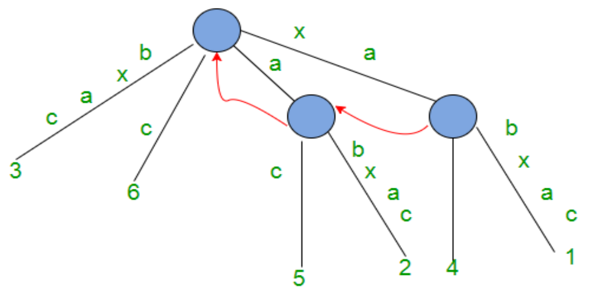


Figure 15 : Suffix links in red arrows

Here red arrow shows the suffix link in a suffix tree. Reason to use suffix links are following:

a) A suffix link can be used to skip the node as it has same suffix hence *keepMatching()* function can be executed in O(1) time.
b) While generating T_{i+1} from T_i, every time we don't need to start from root and traverse down to leaf, we can use suffix link to find the next branch in which we can directly call *keepMatcing()*, hence *contains()* function can also be executed in O(1) time.

So, use of suffix link reduces the overall time complexity of *generateNextOf()* to O(1) time and hence total time complexity to build complete suffix tree .i.e. generating 'm' new tree will take O(m) time.

Now let us look at how to reduce space complexity.

Edge Labelled or Compact Suffix Tree

Consider suffix *s(X)* : "ana$" of string *X* : "ban**ana$**"

Index of *s(X)* in *X* = 4 to 7 (counting 'b' as 1)

So, we can denote *s(X)* as *X[4,7]* or in general any suffix of *X* can be represented as *X[start_index, end_index].*

Definition : *A suffix tree is called* ***Compact Suffix Tree*** *if all the nodes/edges labelled with any string are represented in [start_index, end_index].*

Here is an example of compact suffix tree. This example uses length of string instead of end index which is also fine with respect to our purpose.

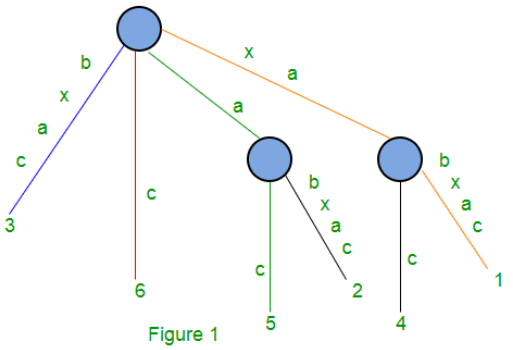


Figure 1

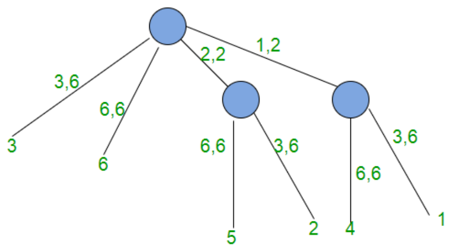
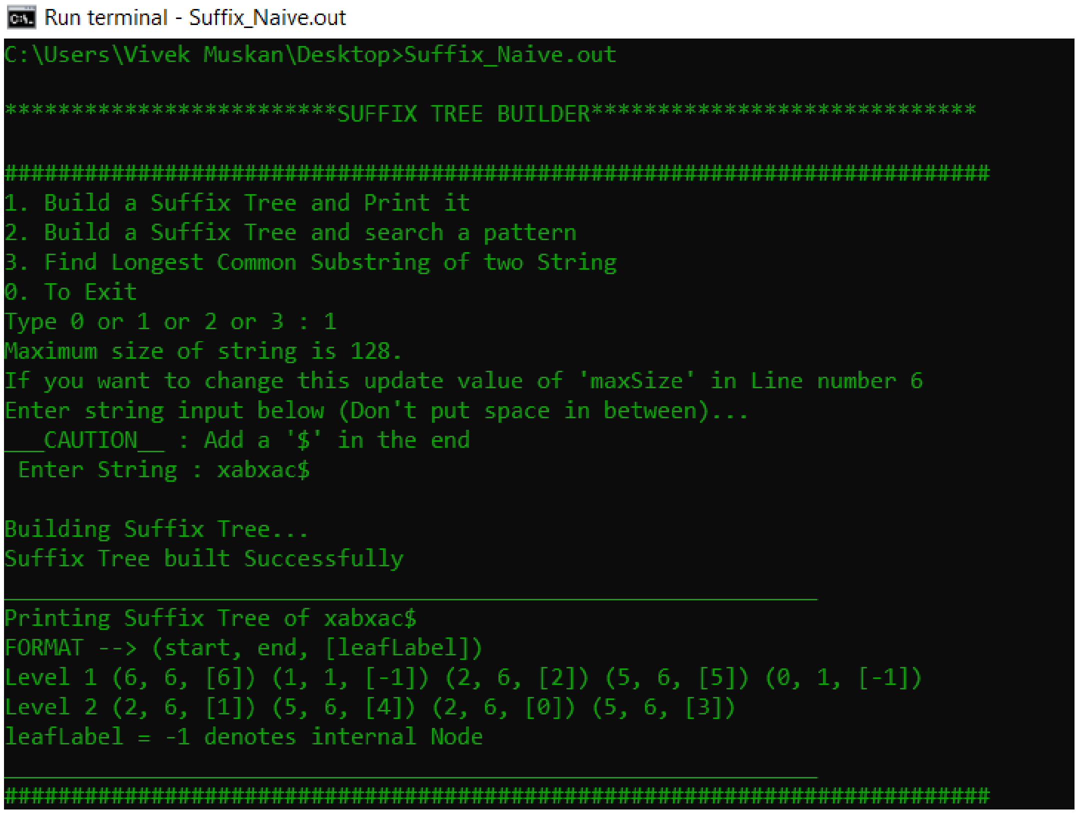


Figure 19 : Suffix tree for string xabxac with edge-label compression
Figure 14 shows same suffix tree without edge-label compression



Run terminal - Suffix_Naive.out

```
C:\Users\Vivek Muskan\Desktop>Suffix_Naive.out

*************************SUFFIX TREE BUILDER*****************************

###########################################################################
1. Build a Suffix Tree and Print it
2. Build a Suffix Tree and search a pattern
3. Find Longest Common Substring of two String
0. To Exit
Type 0 or 1 or 2 or 3 : 1
Maximum size of string is 128.
If you want to change this update value of 'maxSize' in Line number 6
Enter string input below (Don't put space in between)...
___CAUTION__ : Add a '$' in the end
 Enter String : xabxac$

Building Suffix Tree...
Suffix Tree built Successfully
______________________________________________________________
Printing Suffix Tree of xabxac$
FORMAT --> (start, end, [leafLabel])
Level 1 (6, 6, [6]) (1, 1, [-1]) (2, 6, [2]) (5, 6, [5]) (0, 1, [-1])
Level 2 (2, 6, [1]) (5, 6, [4]) (2, 6, [0]) (5, 6, [3])
leafLabel = -1 denotes internal Node
______________________________________________________________
###########################################################################
```

We can see the nodes on every level matches to above figure

Now we can store our main String as global variable and use this representation for all practical purpose. If we look at the space used by every node/edge label is a pair of integers, which uses O(1) space and we have already seen that a suffix tree can have a maximum of m paths from root to leaf hence total space complexity will be O(m) times.

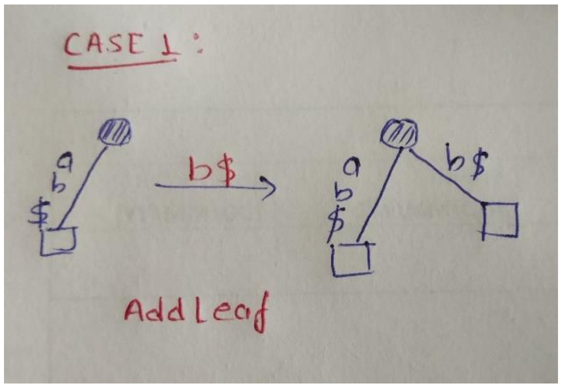
4. My Implementation

My implementation algorithm follows high level McCreight's Algorithm but since suffix links are for predicting the next possible node hence it has a very-very large

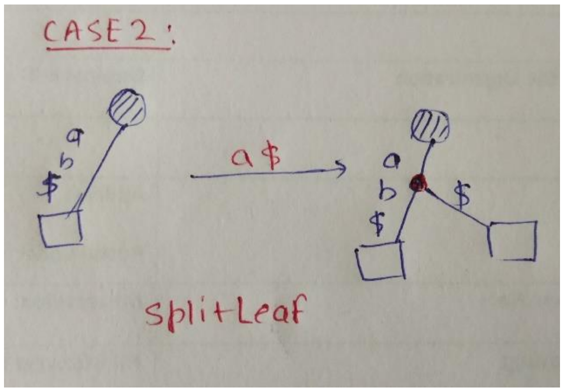
number of cases and sub-cases. Instead I used ASCII code Hash-mapping to predict the next possible node.
I have implemented the compact suffix tree which takes O(m) space.

Algorithm

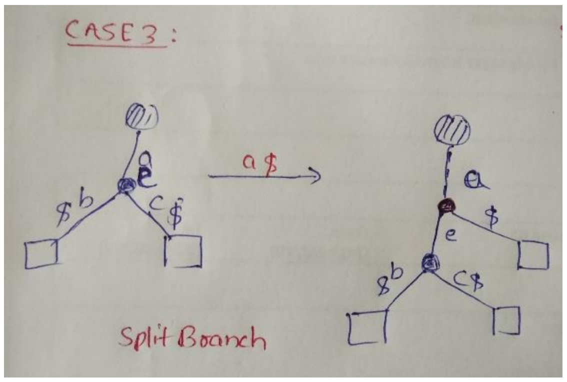
```
buildSuffixTree
    For i = 1:m                     Generating $T_{i+1}$ from $T_i$
            addSuffix(S[i,m]);
    end

end

addSuffix()
    case 1 : No outgoing edge found
            addLeafNode()               O(1)
```



```
    case 2 : Split edge form leaf Node
            addLeafNode()               O(1)
            renameOld()                 O(1)
```



```
    case 3 : Split edge from a Internal Node
            addLeafNode()               O(1)
            renameOld()                 O(1)
            doRotation()                O(1)
```



case 4 : Jump to next Node possible Node recursively
addSuffix()
end

Time Complexity Analysis

Now we are in state Ti i.e. S[1,m], S[2,m], ….S[i,m] are already in the tree. When We call *addSuffix(i+1) :*
It has to fall within any of first 3 cases in first call or successive call **only once,** as we have add only one more suffix S[i+1,m].
Hence Total Complexity of *addSuffix()* will be O(1).

Note : *In 4th case, there is a cost of traversing down the tree until we fall in any of the cases from 1 to 3. It can be made little faster by skipping nodes but still in worst case it can be O(m) times for significantly large m.*
Hence total time complexity of this algorithm will be O(m^2).

Space Complexity

As already discussed above since it is a compact Suffix Tree it will have O(m) space.

5. Application

Main application of suffix tree is in pattern searching, which is vastly used in Genome Science. It is easier to answer questions like :

1. Whether a string S is a substring of T?
2. Whether S is a suffix of T>
3. How many number of times did S occurred in T and at what positions?
4. What is the longest matching between S and T?

So I have implemented two applications which covers all these question.

Pattern Searching

Given any suffix tree and a pattern that has to be searched, it prints all the places of where this pattern can be found. Here is my algorithm to implement this.

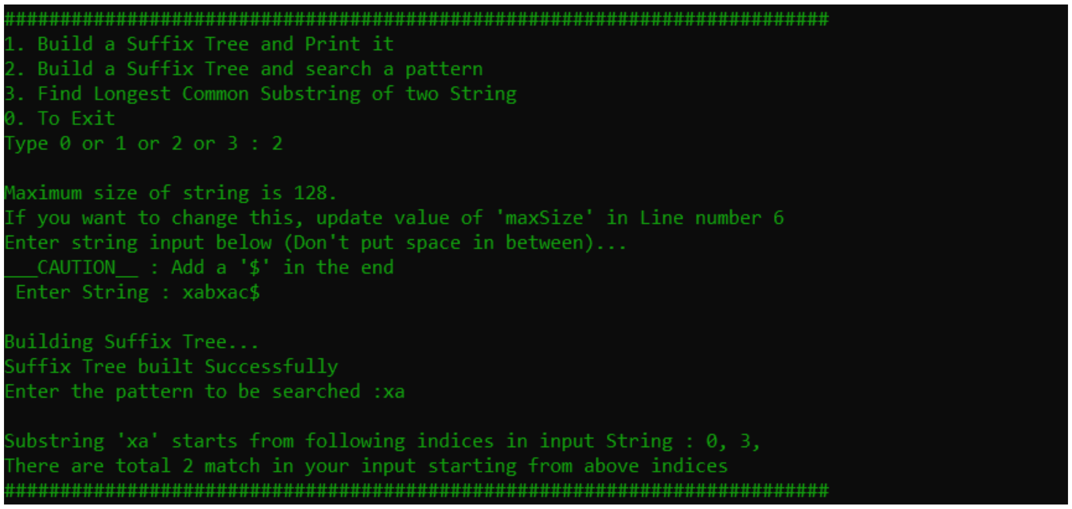
Algorithm

```
patternSearching(P[1,k])
    For i = 1 to k
            H = hashCode(P[1])
            Follow H^th Node
            keepMatching()
            if not Found -> return FALSE
            if Node become empty
                    H_new  = hashCode(P[new])
                    Now Follow H_new4^th  Node
            End if
    End for
    getAllLeafLevel()
    return TRUE;
End
```

Algorithm is simple that I keep on following expected edge until either it did not match(return False) or it matched completely at some node. Then all the leaf coming out of that node will have the same substring so I print all the possible indices.

Time Complexity

It is linear search till k^{th} step then a DFS to generate all possible leaf (let n), so total time will be O(n+k).



```
##########################################################################
1. Build a Suffix Tree and Print it
2. Build a Suffix Tree and search a pattern
3. Find Longest Common Substring of two String
0. To Exit
Type 0 or 1 or 2 or 3 : 2

Maximum size of string is 128.
If you want to change this, update value of 'maxSize' in Line number 6
Enter string input below (Don't put space in between)...
___CAUTION__ : Add a '$' in the end
 Enter String : xabxac$

Building Suffix Tree...
Suffix Tree built Successfully
Enter the pattern to be searched :xa

Substring 'xa' starts from following indices in input String : 0, 3,
There are total 2 match in your input starting from above indices
##########################################################################
```

Gives indices as 3, 0 of 'xa' when searched in 'xabxac$'

Longest Common Substring

If we have two string X and Y of length l_1 and l_2 then form a new string S = X#Y$, where #,$ are special characters to distinguish between them. Idea is that any substring T[i, j] : if i < l_1+1 then it is a part of X for sure

And if j > l_1+1then it is part of Y for sure.

Hence get all the nodes which have such property and find the deepest Node. The label of that node should give maximum common substring.

Algorithm

At first I made a 4-tuple struct (Min, Max, DoesHoldProperty, String) [say MyStruct] which every will be compared at every node.

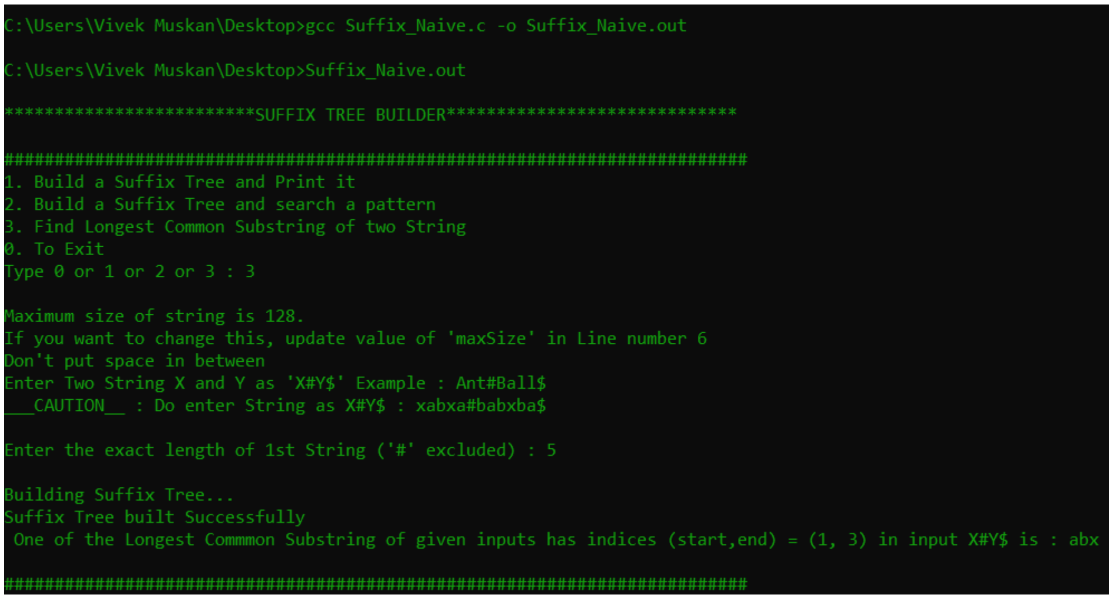
Min, Max : Answers whether property holds?

String : The part of string which is common

```
MyStruct LCS()
    If LeafNode return (LeafIndex, LeafIndex, False, Null)
    Else
        For all child of current node
            LCS()
        End
        Get MyStruct With Max string length over all Children
        OR, Compare()
    Concatenate label of current Node with ans
RETURN
```

Here is How I compared them

```
Compare()
    Min over all Min of Children
    Max over all Max of Child
    OR over all Bool of Children
    MaxLength String
End
```



```
C:\Users\Vivek Muskan\Desktop>gcc Suffix_Naive.c -o Suffix_Naive.out

C:\Users\Vivek Muskan\Desktop>Suffix_Naive.out

************************SUFFIX TREE BUILDER****************************

#######################################################################
1. Build a Suffix Tree and Print it
2. Build a Suffix Tree and search a pattern
3. Find Longest Common Substring of two String
0. To Exit
Type 0 or 1 or 2 or 3 : 3

Maximum size of string is 128.
If you want to change this, update value of 'maxSize' in Line number 6
Don't put space in between
Enter Two String X and Y as 'X#Y$' Example : Ant#Ball$
___CAUTION__ : Do enter String as X#Y$ : xabxa#babxba$

Enter the exact length of 1st String ('#' excluded) : 5

Building Suffix Tree...
Suffix Tree built Successfully
 One of the Longest Commmon Substring of given inputs has indices (start,end) = (1, 3) in input X#Y$ is : abx

#######################################################################
```

LCS of ‘ xabxa#babxba$’ gives ‘abx’ as longest substring with index[1,3]

6. Conclusion

Due to its large application in genetics, it is very much useful with a linear time complexity. Building a suffix tree is mainly a search algorithm in which are mainly improved by McCreight’s Algorithm and Ukkonen’s Algorithm by predicting the best possible direction for search and this is the reason why these algorithm has so many cases and sub-cases.