Generalized Suffix Tree 1 - GeeksforGeeks

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for substring check, searching all patterns, longest repeated substring and built suffix array (All linear time operations).

There are lots of other problems where multiple strings are involved.
e.g. pattern searching in a text file or dictionary, spell checker, phone book, Autocomplete, Longest common substring problem, Longest palindromic substring and More.

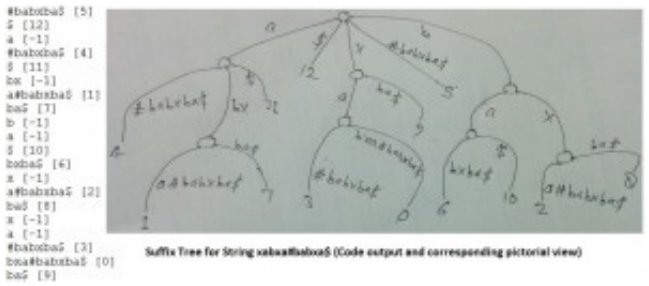
For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.
A suffix tree made of a set of strings is known as Generalized Suffix Tree.
We will discuss a simple way to build Generalized Suffix Tree here for **two strings only**.
Later, we will discuss another approach to build Generalized Suffix Tree for **two or more strings**.

Here we will use the suffix tree implementation for one string discussed already and modify that a bit to build generalized suffix tree.

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y$ where # and $ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then
X#Y$ = xabxa#babxba$
If we run the code implemented at Ukkonen's Suffix Tree Construction – Part 6 for string xabxa#babxba$, we get following output:

(Click to see it clearly)

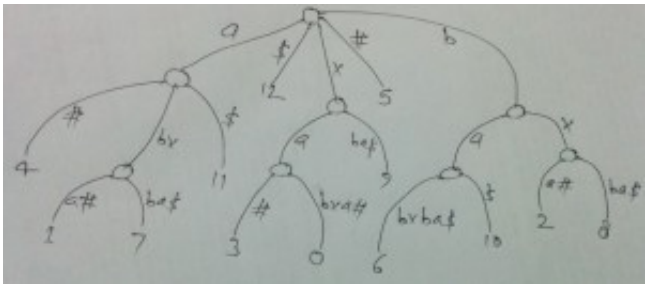


Suffix Tree for String xabxa#babxba$ (Code output and corresponding pictorial view)

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion

corresponding to one string and remove all the later portion. For example, for path labels #babxba$, a#babxba$ and bxa#babxba$, we can remove babxba$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:

(Click to see it clearly)



Below implementation is built on top of original implementation. Here we are removing unwanted characters on path labels. If a path label has “#” character in it, then we are trimming all characters after the “#” in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y$

```
// A C program to implement Ukkonen's Suffix Tree Construction// And then
build generalized suffix tree#include <stdio.h>#include <string.h>#include
<stdlib.h>#define MAX_CHAR 256
struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
       node is connected to its parent node. Each edge will
       connect two nodes,  one parent and one child, and
       (start, end) interval of a given edge  will be stored
       in the child node. Lets say there are two nods A and B
       connected by an edge with indices (5, 8) then this
       indices (5, 8) will be stored in node B. */
    int start;
    int *end;
    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};
typedef struct SuffixTreeNode Node;
char text[100]; //Input string
Node *root = NULL; //Pointer to root node
```

```
/*lastNewNode will point to newly created internal node,
    waiting for it's suffix link to be set, which might get
    a new suffix link (other than root) in next extension of
    same phase. lastNewNode will be set to NULL when last
    newly created internal node (if there is any) got it's
    suffix link reset to new internal node created in next
    extension of same phase. */
Node *lastNewNode = NULL;Node *activeNode = NULL;/*activeEdge is represeted
as input string character
    index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;
// remainingSuffixCount tells how many suffixes yet to// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;
    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;
    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}
int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}
```

```
int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick  (Trick 1). If activeLength is greater
    than current edge length, set next  internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}
void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;
    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;
    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;
    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {
        if (activeLength == 0)
            activeEdge = pos; //APCFALZ
        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);
            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
```

```
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
              is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to curent active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }
                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
                and move on to next phase*/
                break;
            }
            /*We will be here when activePoint is in middle of
```

```
                the edge being traversed and current character
                being processed is not  on the edge (we fall off
                the tree). In this case, we add a new internal node
                and a new leaf edge going out of that new node. This
                is Extension Rule 2, where a new leaf edge and a new
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;
            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;
            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;
            /*We got a new internal node here. If there is any
                internal node created in last extensions of same
                phase which is still waiting for it's suffix link
                reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
                created internal node*/
                    lastNewNode->suffixLink = split;
            }
            /*Make the current newly created internal node waiting
                for it's suffix link reset (which is pointing to root
                at present). If we come across any other internal node
                (existing or newly created) in next extension of same
                phase, when a new leaf edge gets added (i.e. when
                Extension Rule 2 applies is any of the next extension
                of same phase) at that point, suffixLink of this node
                will point to that internal node.*/
            lastNewNode = split;
        }
        /* One suffix got added in tree, decrement the count of
            suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
```

```
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}
void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}//Print the suffix tree as well along with setting suffix index//So tree will be printed in DFS manner//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;
    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);
            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
```

```
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#') //Trim unwanted characters
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}
void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}/*Build the suffix tree and print the edge labels along with suffixIndex.
suffixIndex for leaf edges will be >= 0 and for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);
    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
```

```
    setSuffixIndexByDFS(root, labelHeight);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}// driver program to test above functions
int main(int argc, char *argv[])
{//  strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}
```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```
# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]
```

If two strings are of size M and N, this implementation will take O(M+N) time and space.
If input strings are not concatenated already, then it will take 2(M+N) space in total, M+N space to store the generalized suffix tree and another M+N space to store concatenated string.

Followup:
Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.
We will discuss another approach to build generalized suffix tree soon where we will need only one

unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above