Suffix Tree Application 5 - Longest Common Substring - GeeksforGeeks

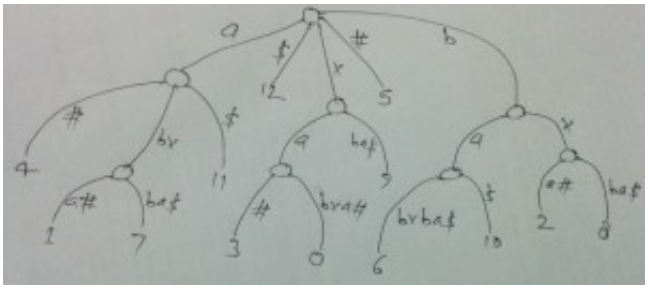
Suffix Tree Application 5 – Longest Common Substring

Given two strings X and Y, find the Longest Common Substring of X and Y.

Naive [O(N*M^2)] and Dynamic Programming [O(N*M)] approaches are already discussed here.
In this article, we will discuss a linear time approach to find LCS using suffix tree (The 5th Suffix Tree Application).
Here we will build generalized suffix tree for two strings X and Y as discussed already at:
Generalized Suffix Tree 1

Lets take same example (X = xabxa, and Y = babxba) we saw in Generalized Suffix Tree 1.
We built following suffix tree for X and Y there:

(Click to see it clearly)



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. Why ??
Because in concatenated string xabxa#babxba$, index of string xabxa is 0 and it's length is 5, so indices of it's suffixes would be 0, 1, 2, 3 and 4. Similarly index of string babxba is 6 and it's length is 6, so indices of it's suffixes would be 6, 7, 8, 9, 10 and 11.

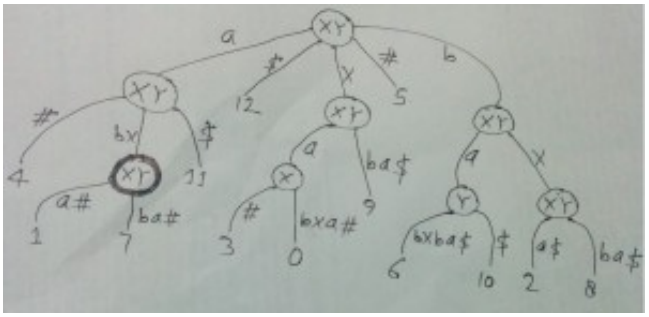
With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from

- both strings X and Y (i.e. there is at least one leaf with suffix index in [0,4] and one leaf with suffix index in [6, 11]
- string X only (i.e. all leaf nodes have suffix indices in [0,4])
- string Y only (i.e. all leaf nodes have suffix indices in [6,11])

Following figure shows the internal nodes marked as "XY", "X" or "Y" depending on which string the

leaves belong to, that they have below themselves.

(Click to see it clearly)



What these “XY”, “X” or “Y” marking mean ?
Path label from root to an internal node gives a substring of X or Y or both.
For node marked as XY, substring from root to that node belongs to both strings X and Y.
For node marked as X, substring from root to that node belongs to string X only.
For node marked as Y, substring from root to that node belongs to string Y only.

By looking at above figure, can you see how to get LCS of X and Y ?
By now, it should be clear that how to get common substring of X and Y at least.
If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

Now we need to find the longest one among all those common substrings.
Can you think how to get LCS now ? Recall how did we get Longest Repeated Substring in a given string using suffix tree already.
The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label “abx” from root to that node is the LCS of X and Y.

```
// A C program to implement Ukkonen's Suffix Tree Construction// Here we build generalized suffix tree for two strings// And then we find longest common substring of the two input strings#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
int size1 = 0; //Size of 1st 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
        //Uncomment below line to print suffix tree
        //print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
```

```
            //Uncomment below two lines to print suffix index
        //   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] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
       // 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);
}
int doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                ret = doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);
                if(n->suffixIndex == -1)
                    n->suffixIndex = ret;
                else if((n->suffixIndex == -2 && ret == -3) ||
                    (n->suffixIndex == -3 && ret == -2) ||
                    n->suffixIndex == -4)
                {
                    n->suffixIndex = -4;//Mark node as XY
                    //Keep track of deepest node
                    if(*maxHeight < labelHeight)
```

```
                    {
                        *maxHeight = labelHeight;
                        *substringStartIndex = *(n->end) -
                                labelHeight + 1;
                    }
                }
            }
        }
    }
    else if(n->suffixIndex > -1 && n->suffixIndex < size1)//suffix of X
        return -2;//Mark node as X
    else if(n->suffixIndex >= size1)//suffix of Y
        return -3;//Mark node as Y
    return n->suffixIndex;
}
void getLongestCommonSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No common substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 7;
    printf("Longest Common Substring in xabxac and abcabxabcd is: ");
    strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 10;
    printf("Longest Common Substring in xabxaabxa and babxba is: ");
    strcpy(text, "xabxaabxa#babxba$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
```

```
    freeSuffixTreeByPostOrder(root);
    size1 = 14;
    printf("Longest Common Substring in GeeksforGeeks and GeeksQuiz is:
");
    strcpy(text, "GeeksforGeeks#GeeksQuiz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 26;
    printf("Longest Common Substring in OldSite:GeeksforGeeks.org");
    printf(" and NewSite:GeeksQuiz.com is: ");
    strcpy(text, "OldSite:GeeksforGeeks.org#NewSite:GeeksQuiz.com$");
    buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Common Substring in abcde and fghie is: ");
    strcpy(text, "abcde#fghie$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Common Substring in pqrst and uvwxyz is: ");
    strcpy(text, "pqrst#uvwxyz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    return 0;
}
```

Output:

```
Longest Common Substring in xabxac and abcabxabcd is: abxa, of length: 4
Longest Common Substring in xabxaabxa and babxba is: abx, of length: 3
Longest Common Substring in GeeksforGeeks and GeeksQuiz is: Geeks, of
length: 5
Longest Common Substring in OldSite:GeeksforGeeks.org and
NewSite:GeeksQuiz.com is: Site:Geeks, of length: 10
Longest Common Substring in abcde and fghie is: e, of length: 1
Longest Common Substring in pqrst and uvwxyz is: No common substring
```

If two strings are of size M and N, then Generalized Suffix Tree construction takes O(M+N) and LCS

finding is a DFS on tree which is again O(M+N).
So overall complexity is linear in time and space.

Followup:

1. Given a pattern, check if it is substring of X or Y or both. If it is a substring, find all it's occurrences along with which string (X or Y or both) it belongs to.
2. Extend the implementation to find LCS of more than two strings
3. Solve problem 1 for more than two strings
4. Given a string, find it's Longest Palindromic Substring

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