Suffix Tree Application 2 - Searching All Patterns - GeeksforGeeks

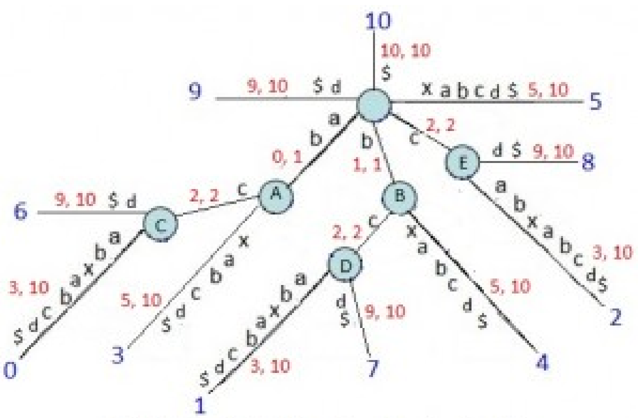
Suffix Tree Application 2 – Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application (Substring Check), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through Substring Check 1st.
In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

Lets look at following figure:



Suffix Tree for String S = abcabxabcd$

This is suffix tree for String “abcabxabcd$”, showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.
Suffix Index of a path tells the index of a substring (starting from root) on that path.
Consider a path “bcd$” in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd$ are at index 7 in string.
Similarly path “bxabcd$” with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcd$ are at index 4.
Similarly path “bcabxabcd$” with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring “b” is at indices 1, 4 and 7
- Substring “bc” is at indices 1 and 7

With above explanation, we should be able to see following:

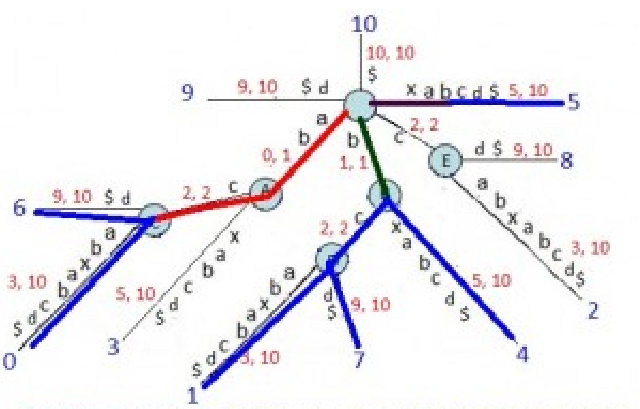
- Substring “ab” is at indices 0, 3 and 6
- Substring “abc” is at indices 0 and 6
- Substring “c” is at indices 2 and 8
- Substring “xab” is at index 5
- Substring “d” is at index 9
- Substring “cd” is at index 8

  …..

  …..

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in [Substring Check]). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don’t fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string



1. Substring abc is found, subtree traversal shows that it is at indices 0 and 6.
2. Substring b is found, subtree traversal shows that it is at indices 1, 4 and 7.
3. Substring xab is found, subtree traversal shows that it is at index 5.

```
// A C program to implement Ukkonen's Suffix Tree Construction// And find all locations of a pattern in string#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; k++)
        printf("%c", text[k]);
}//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)
    {
        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 traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}
int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
    }
    return count;
}
int countLeaf(Node *n)
{
```

```
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}
int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res == -1)  //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                               n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}
void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
```

```
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    return 0;
}
```

Output:

```
Text: GEEKSFORGEEKS, Pattern to search: GEEKS
```

```
Found at position: 8
Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring


Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring


Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring


Text: AAAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
```

```
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring


Text: AAAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring


Text: AAAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring


Text: AAAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M and then if there are Z occurrences of the pattern, it will take O(Z) to find indices of all those Z occurrences.
Overall pattern complexity is linear: O(M + Z).

A bit more detailed analysis

How many internal nodes will there in a suffix tree of string of length N ??

Answer: N-1 (Why ??)

There will be N suffixes in a string of length N.
Each suffix will have one leaf.
So a suffix tree of string of length N will have N leaves.
As each internal node has at least 2 children, an N-leaf suffix tree has at most N-1 internal nodes.
If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have Z-1 internal nodes. A tree with Z leaves can be traversed in O(Z) time.
Overall pattern complexity is linear: O(M + Z).
For a given pattern, Z (the number of occurrences) can be atmost N.
So worst case complexity can be: O(M + N) if Z is close/equal to N (A tree traversal with N nodes take O(N) time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

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