Ukkonen's Suffix Tree Construction – Part 6

This article is continuation of following five articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5

Please go through Part 1, Part 2, Part 3, Part 4 and Part 5, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string "abcabxabcd" where we went through all phases of building suffix tree.
Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of Part 5 article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.
There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (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 (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#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) {
    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
        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)
    {
        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, "abc"); buildSuffixTree();
//  strcpy(text, "xabxac#");    buildSuffixTree();
//  strcpy(text, "xabxa");  buildSuffixTree();
//  strcpy(text, "xabxa$"); buildSuffixTree();
```

```
    strcpy(text, "abcabxabcd$"); buildSuffixTree();
//  strcpy(text, "geeksforgeeks$"); buildSuffixTree();
//  strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
//  strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}
```

Run on IDE

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous Part 5 article):

```
$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]
```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, KMP otherwise
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- Linear Time Suffix Array Creation

The above basic problems can be solved by DFS traversal on suffix tree.
We will soon post articles on above problems and others like below:

- Build Generalized suffix tree
- Linear Time Longest common substring problem
- Linear Time Longest palindromic substring

And More.

Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string “AABAACAADAABAAABAA$” on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3.
   Following are the rules applied on five consecutive extensions in some Phase i (i > 5), which ones are

valid:

A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3

B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2

C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3

D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1

E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2

F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3

3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

References:

http://web.stanford.edu/~mjkay/gusfield.pdf

Ukkonen's suffix tree algorithm in plain English

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

10 Comments Category: Strings Tags: Pattern Searching

Related Questions:

- Shortest Superstring Problem
- Shortest Common Supersequence
- How to design a tiny URL or URL shortener?
- Remove spaces from a given string
- Online algorithm for checking palindrome in a stream
- Recursively print all sentences that can be formed from list of word lists
- Check if a given sequence of moves for a robot is circular or not
- Find the longest substring with k unique characters in a given string

10 Comments GeeksforGeeks Priyanka Khire

Recommend Share Sort by Newest

Join the discussion…

Mohammad Adi · 22 days ago
Have you tried running this with the input string "mississippi"? I dont think it works.
· Reply · Share ›

Adi ↗ Mohammad Adi · 22 days ago
Never mind...I may have misunderstood the output.
· Reply · Share ›

melonhead · a month ago
What license are you willing to release this code under? MIT, maybe...?
· Reply · Share ›

Mission Peace · a month ago
https://youtu.be/aPRqocoBsFQ Video on suffix link
· Reply · Share ›

Steve · 3 months ago
Why do we need "activeNode != root" when setting the suffix link in rule 3?
· Reply · Share ›

Anurag Singh ↗ Steve · 3 months ago
Probably that condition can be removed and it should not cause any error.
But to be sure, need to test all scenarios to make sure it doesn't error out in any case.
You may try out and see if all good.
I'll check it later and update.
· Reply · Share ›

DP · 5 months ago
In extendSuffixTree method, in the else condition when a match is found, activeLength is incremented and shouldn't it return here instead of break?
When break is executed, is comes out of while loop and remainder, activeLength is decremented

decremented.

• Reply • Share ›

DP ➚ DP • 5 months ago

IGNORE my comment. I'm wrong. The code that decrements remainder and activeLength in inside the while loop.

• Reply • Share ›

Anurag Singh ➚ DP • 5 months ago

Match found means the character you are trying to add is present in tree, so the phase ends there (APCFER3).
So we use break to come out of loop (as no more extension needed on that phase) and so we come out of extendSuffixTree function itself.
Control goes back to buildSuffixTree where it calls extendSuffixTree for next character to be added (the next phase processing).
Look at APCFER3 (activePoint change for extension rule 3) in Ukkonen's Suffix Tree Construction – Part 3 to understand it well.

• Reply • Share ›

DP ➚ Anurag Singh • 5 months ago

Sorry, i missed out that the loop closes at the end of the function. I realized it as soon as i posted the comment and tried to delete my comment but it failed.

• Reply • Share ›