Suffix Trees

Akhil K, *UE15CS311*, *Student at Department of CSE, PES University*

***Abstract* — We consider the problem of using suffix trees to solve string-matching problems. Existing string matching algorithms parse the *pattern* instead of *text*. Implementation of suffix trees is written in C. The suffix tree contains user defined "*nodes*" (implemented with structures in C). The time complexity of this algorithm is in the order of O(n) where n is the length of the *pattern* to be processed. Suffix trees are helpful not only in string matching, but also in indexing documents. This makes it ideal to solve our problem. Also, questions [1] followed by creation of suffix trees are also solved. We are using Aesop Tales [2] as our input where each tale is a document.**

I. INTRODUCTION

EVERY string matching algorithm until now was designed around the *pattern* instead of *text*. The disadvantage of this is that it increases running time in the order of the *text*, not pattern. If you want to search for a small *pattern* in a lengthy *text*, for example, legacy algorithms process the entire *text* to search for your *pattern*. But, by processing *text* beforehand (in the algorithm), suffix trees need to process only the *pattern* during runtime. Even in the worst-case scenario, the time complexity doesn't go beyond that of legacy string matching algorithms.

Suffix trees can be implemented in many ways but we are using the *tree* data structure using *structures* in C. It is a more intuitive way of building a suffix tree instead of abstract algorithms such as suffix arrays, etc.

II. SUFFIX TREE CREATION

A. Data Structure in use

The data elements in a node is given below.

```
struct STree{
   char ele;
   struct STree ** children;
   int n_children;
   int* tale;
   int* tale_index;
   int n_tale;
};
```

Submission date: 17th October 2017
Akhil K is in Department of CSE, PES University, Bangalore, KA 560085, India (e-mail: blitzkrieg.akhil@gmail.com).

ele is the character on which the pointer reaches that particular node, *children* is an array containing pointer to a node's children. *n_children* is the number of children for a node. *tale* is the tale number in which that pattern occurred. For a particular node, if the pattern traversal doesn't terminate at that node or its ancestors, there will be an entry in *tale*. This indicates that a substring of pattern having same characters as that of the line of traversal from the root node to the current node is at that tale number.

Like *tale*, *tale_index* keeps a track of the index of tale at which that substring match occurred. This may not make sense now but will be used later to solve the questions[1].

Taking one document at a time and parsing it, we create suffix trees. This document is passed on as a string S of length L. Each document is padded with a '*$*' symbol to indicate end of string. For this string S, we find all possible suffixes and work on the tree for each suffix. Root node contains # as its *ele* and each termination node is represented by a *$*.

B. Method of Traversal

1. For a node, the parser checks if it has a node. If it doesn't and the current element of pattern is *$* (which indicates end of pattern), it creates a '*$*' node which acts as the terminal node.

```
if(root->children==NULL){
        if(suffix[j]=='$'){
            // Create terminal node
            root->n_children++;
            root->children=(struct STree**)realloc(root-
>children,sizeof(struct STree*)*root->n_children);
            struct STree* dol=malloc(sizeof(struct STree));
            dol->ele='$';
            dol->children=NULL;
            dol->n_tale=0;
            dol->n_children=0;
            root->children[root->n_children-1]=dol;
        }
```

2. If the current element isn't a terminal character (*$*) and the current node has no children, two child nodes are created. One is $ (terminal character) and the current character of *suffix*. The parser now enters the node of the current character.

```
        else{
            //Create a $ node
            root->n_children++;
            root->children=(struct STree**)realloc(root-
>children,sizeof(struct STree*)*root->n_children);
```

```
            struct STree* dol=malloc(sizeof(struct STree));
            dol->ele='$';
            dol->children=NULL;
            dol->n_tale=0;
            dol->n_children=0;
            root->children[root->n_children-1]=dol;
            //Create a node suffix[j]
            root->n_children++;
            root->children=(struct STree**)realloc(root-
>children,sizeof(struct STree*)*root->n_children);
            struct STree* new=malloc(sizeof(struct STree));
            new->ele=suffix[j];
            new->children=NULL;
            new->n_tale=1;
            new->n_children=0;
            root->children[root->n_children-1]=new;
            new->tale=malloc(sizeof(int));
            new->tale[new->n_tale-1]=tale;
            new->tale_index=malloc(sizeof(int));
            new->tale_index[new->n_tale-1]=n-i+j;
            root=new;
          }
        }
```

3. In case there are child nodes for the current node, the parses checks if there is a character match in its children (*ele* of every node) with the current character of *suffix* and enters the node with that character.

```
        else{
          int present=0;
          for(k=1;k<root->n_children;k++){
            //Check for character in children
            if(root->children[k]->ele==suffix[j]){
              present=1;
              if(suffix[j]!='$'){
                root=root->children[k];
                root->n_tale++;
                root->tale=realloc(root-
>tale,sizeof(int)*root->n_tale);
                root->tale_index=realloc(root-
>tale_index,sizeof(int)*root->n_tale);
                root->tale_index[root->n_tale-1]=n-i+j;
                root->tale[root->n_tale-1]=tale;
                break;
              }
            }
          }
```

4. If the node with current character of *suffix* isn't present, we create a new node with the current character of *suffix* and enter that node.

```
          if(present==0){
            root->n_children++;
            root->children=(struct STree**)realloc(root-
>children,sizeof(struct STree*)*root->n_children);
            struct STree* new=malloc(sizeof(struct STree));
            new->ele=suffix[j];
            new->children=NULL;
            new->n_tale=1;
            new->n_children=0;
            root->children[root->n_children-1]=new;
            new->tale=malloc(sizeof(int));
            new->tale[new->n_tale-1]=tale;
            new->tale_index=malloc(sizeof(int));
            new->tale_index[new->n_tale-1]=n-i+j;
            root=new;
          }
        }
      }
```

5. For every *suffix*, we return to the root node to start parsing the next *suffix*.

```
      root=masterroot;
    }
    root=masterroot;
    return root;
}
```

III. QUESTION 1

For Question 1, we have to find all occurences of a query string along with its context (the sentence it is in) and print it.

traverse_q1 traverses through the suffix tree.

```
void traverse_q1(struct STree* root,char query[],int
query_char){
  int i;
```

1. If we have successfully traversed through the entire query-string, we find context of every match.

```
  if(query[query_char]=='\0'){
    for (i=0;i<root->n_tale;i++){
      findSentence(root,root->tale[i],root->tale_index[i]);
    }
  }
  else{
```

2. Else, we loop through the children of the current node to check for a match in the current character of query-string with *ele* of the nodes.

```
    int flag=0;
    for (i=1;i<root->n_children;i++){
      if(root->children[i]->ele==query[query_char]){
        root=root->children[i];
        query_char=query_char+1;
        flag=1;
        break;
      }
    }
```

3. If we find a match, we go to that node.

```
    if(flag){
```

```
        //Continue traversal
        traverse_q1(root,query,query_char);
    }
```

4. If we don't find a match, we terminate traversal.

```
    else{
        //Traversal has fallen off the STree
        printf("No results found\n");
    }
  }
}
```

findSentence finds the tale with the respective tale number and passes it on to *findContext* with the index at which it is found. The function declaration of *findSentence* is given below.

void findSentence(struct STree root,int tree_tale,int tree_tale_index);*

findContext finds the sentence around that index and prints it (When it encounters a double space on both sides, it stops printing characters). The function declaration of *findContext* is below.

void findContext(char string[],int index);

IV. QUESTION 2

For Question 2, we have to find the first occurrence of the query-string in each document. In case we don't, we have to find the first occurrence of the longest substring of the query-string in the document (having atleast 1 character).

traverse_q2 traverses through the suffix tree.

```
void traverse_q2(struct STree* root,char query[],int
query_char,short int atleast_1){
  int i;
```

1. If we have successfully, traversed through the query-string, we find context of the first match of every document.

```
  if(query[query_char]=='\0'){
    int* buff=calloc(500,sizeof(int));
    for (i=0;i<root->n_tale;i++){
      if(buff[root->tale[i]]==0){
        findSentence(root,root->tale[i],root-
>tale_index[i]);
        buff[root->tale[i]]=1;
      }
    }
    free(buff);
  }
```

2. Else, we loop through the children of the current node to check for a match in the current character of query-string with *ele* of the nodes.

```
  else{
    int flag=0;
    for (i=1;i<root->n_children;i++){
      if(root->children[i]->ele==query[query_char]){
        root=root->children[i];
        query_char=query_char+1;
        atleast_1=1;
        flag=1;
        break;
      }
    }
```

3. If we find a match, we go to that node.

```
    if(flag){
      //Continue traversal
      traverse_q2(root,query,query_char,atleast_1);
    }
```

4. If we don't, we go to the parent node of the node where match didn't occur and find the context of the first match of this *substring* (parsing root node till parent of current node will be a *substring* of parsing from root node till current node).

```
    else{
      //Traversal has fallen off the STree
      if(atleast_1){
        int* buff=calloc(500,sizeof(int));
        for (i=0;i<root->n_tale;i++){
          if(buff[root->tale[i]]==0){
            findSentence(root,root->tale[i],root-
>tale_index[i]);
            buff[root->tale[i]]=1;
          }
        }
        free(buff);
      }
```

5. If we don't find atleast a character matching, we terminate traversal.

```
      else{
        printf("Result not found\n");
      }
    }
  }
}
```

findSentence and *findContext* have the same meaning as the functions of the same name in Part IV (Question 2).

V. QUESTION 3

For Question 3, we have to list the tales by their relevance. Relevance definition:

1. If the query EXACTLY matches a substring in a document, it gets the high relevance.
2. More the number of query matches, higher is the relevance.
3. If query doesn't match, individual words in the query are checked.
4. More number of matched words in a document, higher the relevance.
5. Partial word matches get zero relevance.

traverse_q3_wrapper manages the ordering of documents based on relevance.

```
void traverse_q3_wrapper(struct STree* root,char query[]){
   struct STree* main=root;
   short int mode_status=1;
```

Mode 1: In this mode, we check if the ENTIRE query can be matched as it has more relevance (refer to the relevance definition).

```
   //Mode 1
   if(mode_status==1){
      main=traverse_q3(main,query,0);
```

If there is no query match, we shift to Mode 2.

```
      if(main==NULL){
         mode_status=2;
      }
```

If there is a query match, we get all the matches in all the documents. Later, we assign ranks to these documents based on number of occurences of query string in it. And then print the tale numbers by most relevant tales.

```
      else{
         int result[main->n_tale];
         int i;
         for(i=0;i<main->n_tale;i++){
            result[i]=main->tale[i];
         }
         int n_result=sizeof(result)/sizeof(result[0]);
         struct rank var;
         var.n=0;
         var=assign_rank(result,var,n_result);
         //Sort by rank
         quickSort(var.tale,var.occurences,0,var.n-1);
         //Print by rank
         printf("Tale number of most relevant tales\n");
         for(i=var.n-1;i>=0;i--){
            printf("Tale %d with occurences
>%d\n",var.tale[i],var.occurences[i]);
         }
      }
```

Mode 2: In this mode, we check for individual words in the query and then rank each document's relevance based on the sum of total number of occurences of each word in a document. And then print the tale numbers by most relevant tales.

```
   //Mode 2
   if(mode_status==2){
      struct rank var;
      var.n=0;
      char* word = strtok(query," ");
      int i;
      main=root;
      while (word != NULL){
         main=traverse_q3(main,word,0);
         int result[main->n_tale];
         result[0]=-1;
         for (i=0;i<main->n_tale;i++){
            result[i]=main->tale[i];
         }
         if(result[0]!=-1){
            int n_result=sizeof(result)/sizeof(result[0]);
            var=assign_rank(result,var,n_result);
         }
         word = strtok(NULL, " ");
         main=root;
      }
      //Sort by rank
      quickSort(var.tale,var.occurences,0,var.n-1);
      //Print by rank
      printf("Tale number of most relevant tales\n");
      for(i=var.n-1;i>=0;i--){
         printf("Tale %d with occurences
>%d\n",var.tale[i],var.occurences[i]);
      }
   }
}
```

quicksort[3] is a sorting function.

traverse_q3 traverses through the suffix tree. It exhibits the same functionality as *traverse_q1* encountered in Part III (Question 1). Only change is the name of the function and *traverse_q3* returns the node at which traversal was completed successfully.

```
struct rank{
   int* tale;
   int* occurences;
   int n;
};
```

rank is a user-defined datatype. *tale* contains the tale number of a document. *occurences* contains the number of occurences of a *string* in that particular *tale*.

assign_rank assigns ranks to documents (tales) based on the number of occurences of *string* in each document and returns a variable of type *struct rank*.

```
struct rank assign_rank(int tales[],struct rank var,int n){
  int n_tales=n;
  int i;
  int index=0,j;
  for (i=0;i<n_tales;i++){
```

If *tales[i]* is not in *var*, we add it to *var*.

```
    if(!in_tales(tales[i],var)){
      if(var.n==0){
        var.tale=(int*)malloc(sizeof(int)*(var.n+1));
        var.occurences=(int*)malloc(sizeof(int)*(var.n+1));
      }
      else{

        var.tale=(int*)realloc(var.tale,sizeof(int)*(var.n+1)
        );
        var.occurences=(int*)realloc(var.occurences,sizeof(
        int)*(var.n+1));
      }
      var.tale[var.n]=tales[i];
      var.occurences[var.n]=1;
      var.n++;
    }
```

If *tales[i]* is present in *var*, we increment *var*'s number of occurences for that tale by 1.

```
    else{
      for(j=0;j<var.n;j++){
        if(var.tale[j]==tales[i]){
          index=j;
          break;
        }
      }
      var.occurences[index]++;
    }
  }
  return var;
}
```

in_tales is an utility function which returns 0 or 1. It returns 0 if the element is not present in the variable *var*. If present, it returns 1. The function declaration of *in_tales* is given below.

```
short int in_tales(int ele,struct rank var);
```

VI. RELATED WORK

Existing work that focus on Suffix Trees:

1. Introduction to Suffix Trees and Its Many Applications (http://www.inf.fu-berlin.de/lehre/WS02/ALP3/material/sufficTree.pdf)
2. On-line construction of Suffix Trees (https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf)
3. Suffix Trees and their Applications (https://web.stanford.edu/~mjkay/Maass.pdf)

VII. CONCLUSION

This document proposes a *text* based processing in the form of Suffix Trees to solve variable running time for different *patterns* as input.

ACKNOWLEDGMENT

I am grateful to Prof. Channa Bankapur, Department of CSE, PES University for his lectures on Suffix Trees and helping me understand it.

REFERENCES

[1] Assignment - 2 on Suffix Trees - UE15CS311
Available: https://docs.google.com/document/d/1yw0Ou4BgEsES6ZoJhoA1Ln_hOQwag3gKR57DB6s8Ogw/edit#heading=h.gq2uyc3i0c4e

[2] Aesop Tales.txt
Available: https://drive.google.com/file/d/0B-E2SGzqkISrSy1mdm5TUEN5NUk/view

[3] QuickSort from Geeks for Geeks
Available: http://www.geeksforgeeks.org/quick-sort/