Final Project Report
Topic: Suffix Array Induced search vs Ukkonen's Algorithm for string searching

Participants:
Sathyanarayanan Vaithianathan
Sahil Shetty
Sarthak Mishra

Basics

What is string-searching?

String-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

Why string-searching algorithms?

String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems. It helps in performing time-efficient tasks in multiple domains. These algorithms are useful in the case of searching a string within another string. String matching is also used in the Database schema, Network systems etc.

What is a suffix array?

Given a text T[0…n] over alphabet ∑ of size σ, where the last character $ of T is lexicographically smaller than any other character and occurs in T only once. A suffix Ti of T is a substring of T[i, i+1,…n].
An array of these substrings are called as a suffix array.

Introduction

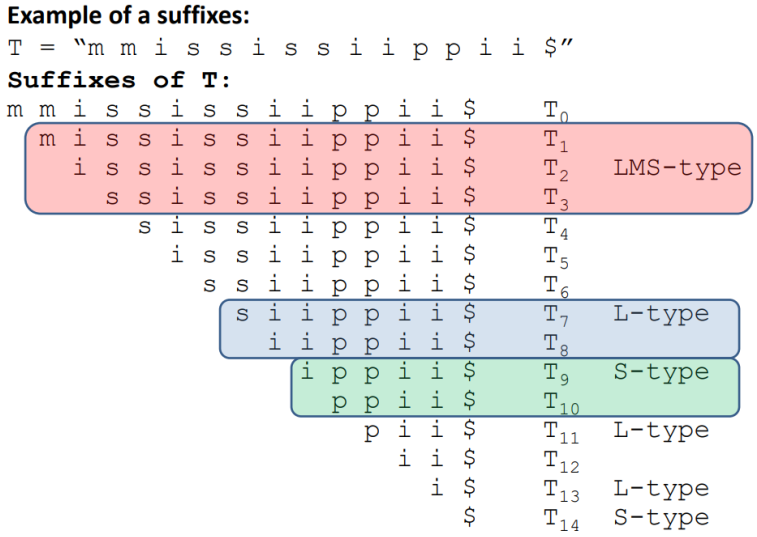
The basic M.O of this report is to show two ways in which we can build suffix arrays. Using which we can implement a string match/search. We will be using two algorithms namely Suffix Array Induced Search(SAIS) and Ukkonen's Algorithm, to build suffix arrays.
Code implementation of both of the algorithms are done and we will be showing the difference in time complexity as well as with real examples to show the same.

Suffix Array by Induced Sorting(SAIS)

- The linear suffix array construction by almost pure Induced-Sorting was first implemented in 2009 by Ge Nong , sen Zhang and Wai Hong Chan.
- This algorithm does suffix array creation in near linear time. Other approaches had been having factorial, squared or cubed time complexities but the get it to near linear time was a breakthrough.

Some terminologies used in SAIS before jumping into the intuition:

- L-Type suffix: A suffix T_i of T is called L-type if it is lexicographically greater than the suffix T_{i+1}, we denote this relation as $T_i > T_{i+1}$
- S-Type Suffix : A suffix is of S-type if $T_i < T_{i+1}$, T_i is alphabetically smaller than T_{i+1}
- Leftmose S-Type (LMS) : A suffix T_i is called Leftmost S-type, LMS (T*), if Ti is S-type, and suffix T_{i-1} is L-type: $T_{i-1} > T_i < T_{i+1}$



- LMS Substring: A substring T[i…j] of T is called LMS-Substring, or in short, T* substring, if T_i and T_j are LMS-suffixes, and none of the suffixes starting between i and j are LMS-suffix; $ is a T* substring of size one,

- c-Bucket: Character bucket, placing suffixes starting with the same character c in their corresponding character buckets.

Intuition

- We first find all the L-type and S-type suffixes and demarcate them in an array.
- We find all the LMS substrings among all the S & L type suffixes.
- Make the suffixes into alphabetical order. And how? Next steps.
- Here comes the concept of induced sorting.
- Then we get the range of entries in the suffix array with the same character bucket called c-bucket. This can be done in linear time.
- Using 3 SAIS techniques we induce sort the LMS substrings first.
- This will basically sort the suffix arrays in a way.
- Now we give names/ short strings to each LMS substring.
- Create a condensed string with the short names
- Recursive call by feeding this new array to the algorithm.

Pseudocode

1. **Induce sort LMS-substrings**
2. **Build T_1:**
    1. **Name LMS substrings**
    2. **Build T_1**
3. **Recursive call on T_1 and SA_1**
4. **Place positions of LMS-substrings into SA using SA_1**
5. **Induce sort suffixes (similar to Step 1)**

Explanation of pseudocode:

Input: Integer array T (text) , integer array SA (Suffix array inititalized to -1; passed by reference)

Output: calculated Suffix array SA
Some important aspects of the pseudocode and the techniques used:

- Before we move to the induced sort of substring in the step 1 of the pseudocode we need to get the ranges of all the S-type, L-Type and LMS strings. Also placing the suffixes in their corresponding c-buckets.
- We need to get all the sorted range array as well.
- The major part of the pseudocode is the build of the T_1 and SA_1.
- There are 3 techniques used to induce sort the LMS substrings and they are (Step 1_:
    - Scan t from Right-to-Left and put positions of all LMS-substrings into the ends of their c-buckets
    - Scan SA from Left-to-Right, and for each SA[i], if TSA[i]-1 is of L-type, put (SA[i]-1) into the current unoccupied front of the corresponding L-bucket (filling L-bucket of c-bucket from left to right)
    - Scan SA from Right-to-Left, and for each SA[i], if TSA[i]-1 is of S-type, put (SA[i]-1) into the current end of the corresponding S-bucket (will overwrite LMS positions) (filling S-bucket of c-bucket from right to left)

- Step 2 substep 1 : Some rules of name assignment are”
    - Compare only two consecutive LMSs
    - If same characters and type, assign same name
    - If different, assign the next name
- Step 2 substep 2 : To create the shortened string T1 : Scan N from Left-to-Right and fill in T1 with the names of LMSs as they occur in N

Runtime analysis:

- Each step takes O(n) time
- Formation of T and SA takes O{n) time because we are using index ranges and not really moving characters around.
- In every recursive call because of the shortened names we get nearly half the string.

  $O(n) + O(n/2) + O(n/4) + \ldots + O(1) = O(n + n/2 + n/4 + \ldots + 1) = O(2n) = O(n)$

Ukkonen's Algorithm

- It is a linear-time, "on-line" algorithm for constructing suffix trees.
- Proposed by Esko Ukkonen in 1995
- It begins with an implicit suffix tree that has the first string of the string.
- It then applies rules and techniques proposed by Ukkonen to subsequently add characters to the tree until its complete.

Some terminologies used in Ukkonen's

- Rule extensions:
  1. **Rule 1**: If the path from the root labeled S[j..i] ends at leaf edge (i.e. S[i] is the last character on the leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.
  2. **Rule 2**: If the path from the root labeled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
     A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.
  3. **Rule 3**: If the path from the root labeled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and the next character is s[i+1] (already in the tree), do nothing.

- Tricks used:
    - Edge label compression where instead of using actual characters we keep the index range of each edge.
    - Global end: We are going to make the lat index of some of the edges for example : [3,end], where the end means the current phase.
    - Skip count: When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if the number of characters on the edge is less than the number of characters we need to travel.
    - Stop processing of any phase as soon as Rule 3 applies.

- Suffix Link: For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.

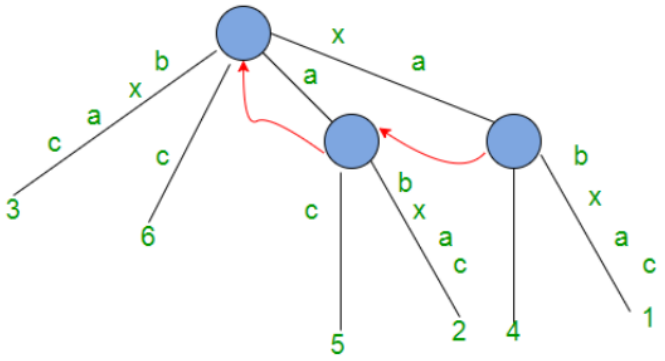


Figure 15 : Suffix links in red arrows

- Active points: Could be root node or any internal node. For any extension of phase from one to the other the active point needs to reset. For this we have 3 variables named: activeNode, activeLength and activeEdge.

Intuition:

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.

- Each edge is labeled with a nonempty substring of S.
- No two edges coming out of the same node can have edge-labels beginning with the same character.

Pseudocode

```
function insert(T, string, start_index, length){
    i = start_index
    while(i < length)
        if T.arr[string[i]]) is NULL
            T.arr[string[i]] = new node()
            T = T.arr[string[i]]
            while(i < length)
                T.label = T.label+string[i]
                i = i+1
            endwhile
            return
        endif

        j=0, x=i
        temp = T.arr[string[i]]
        while j < temp.label.length and i < length && temp.label[j] = string[i]
            i = i+1
            j = j+1
        endwhile

        if i = length
            return
        endif

        if j = temp.label.length
            cnt = 0
            for k = 'a' to 'z'
                if temp.arr[k] = NULL
                    cnt = cnt+1
                endif
            endfor
            if cnt = 27
                while i < length
                    temp.label = temp.label + string[i]
                    i = i+1
                endwhile
            else
                T = temp
            endifelse
```

```
        else
            newInternalNode = new node()
            k=0
            while k < j
                newInternalNode.label = newInternalNode.label + temp.label[k]
                k = k+1
            endwhile
            newInternalNode.arr[string[j]] = new node()
            while k < temp.label.length
                newInternalNode.arr[string[j]].label+=temp.label[k]
                k = k+1
            endwhile
            for k = 'a' to 'z'
                newInternalNode.arr[string[j]].arr[k] = temp.arr[k]
            endfor
            T.arr[string[x]] = newInternalNode
            T = T.arr[string[x]]
        endifelse
    endwhile
}

function makeSuffixTree(string, length){
    Trie T
        string = concatenate(string, "{")
    for i = 0 to length
        insertInTrie(T, string, i, length)
}
```

Pseudocode explanation

Input: String T.
Output: Suffix tree

- We create the initial implicit suffix tree T_i for each prefix S[1…i] of S (length m).
- It first builds T1 using 1st character, then T2 using 2nd character, then T3 using 3rd character, …, Tm using mth character.
- Create the subsequent suffix tree on top of T_1. A true suffix tree for S is built from T_m by adding the sentinel character $.
- Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)
- Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]
- In extension j of phase i+1, the algorithm gets the end of the path from the root labeled with substring S[j..i].

- It then extends the substring by adding the character S(i+1) to its end (if it is not there already). In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in the tree due to previous phase i. We just need to add S[i+1]th character in the tree (if not there already).
- In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in the tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
- In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in the tree (if the character is seen for the first time so far). If so, we just add a new leaf edge with label S[i+1].
- Suffix extension is all about adding the next character into the suffix tree built so far.

Runtime analysis:

At any point of time, Ukkonen builds a suffix tree for the characters it has seen so far so this is termed as on-line property. The time taken for this is O(m) where m is the length of the string.

Space Analysis

1. Suffix tree occupies less space than the trie because we use one leaf for each position in the string which takes O(n).
2. Every internal node in the suffix tree has at least two children
3. Every edge uses O(1) space since we are compressing the space by not storing the characters along the edges.

Findings:

1. Therefore, the number of internal nodes is about equal to the number of leaves.
2. Number of edges is equal to number leaves, thus space is O(1)

Hence the algorithm uses linear space to construct a suffix tree.

Comparison of runtimes for test cases

We ran test cases for both the algorithms for finding a pattern in a given string. We then compared the time taken by each of the algorithms to find out the given pattern in a larger text.
For SAIS we used a binary search after creating the suffix array and for the Ukkonen's algorithm we used DFS to find the occurrences of the pattern.

Test case	SAIS (in ms)	Ukkonen's(in ms)
t00.in	0.4530	0.2666
t01.in	0.4620	0.2677
t02.in	0.3820	0.2123
t03.in	0.4980	0.3485
t04.in	0.3540	0.1778
t05.in	1.0010	0.8193

Difference of time complexities of SAIS and Ukkonen's

	Suffix Array	Suffix Tree
Time Complexity (When Alphabet size is finite and small)	O(N)	O(N)
Space Complexity N is the number of characters in the String	O(N)	$O(N^2)$ for ST O(N) for CST
Search Time m - pattern length n - String length K number of times the pattern occurs	O(m + log n + K)	O(m)

Real world applications of string search algorithms

- We still need to search enormous corpuses of text (Google Search)
- Algorithms for finding long repeated substrings or patterns can be useful for detecting plagiarism.
- Finding all occurrences of a particular substring in some huge corpus.
- We are made out of strings over a particular finite alphabet GATC (gene). String search is a central tool in computational biology.

Deliverables

1. Github Link for Ukkonen's Algorithm
2. Ukkonen Algorithm Playground
3. Github Link for SAIS Algorithm

References

- http://www14.in.tum.de/lehre/2018WS/ada/07_SuffixTrees2.pdf
- Ukkonen’s geek for geeks : https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/?ref=gcse
- SAIS algorithm IEEE paper : https://ieeexplore.ieee.org/document/4976463
- Tushar Roy : https://www.youtube.com/watch?v=aPRqocoBsFQ
- Elena Harris: https://www.youtube.com/watch?v=yb0Os_MTU_4