Project Report

CS 764 Fall 2017

Building a Suffix Tree/Array Index on String Attributes in Quickstep

Arpith Neelavara
neelavara@wisc.edu

Sudarshan Avish Maru
smaru@wisc.edu

1. Introduction

Large text datasets are common in a number of applications. For example, these are extensively used in Digital libraries, Bioinformatics etc. In genomics, in a large database of gene sequences, a particular gene sequence may be of great interest to the researchers. So efficiently searching for such a sequence in a database is an important problem. Researchers have attempted to solve these problems by building an index over such databases using data structures such as B-trees, Suffix arrays, Suffix trees etc. In this project, we investigated ways to do so using suffix arrays and suffix trees. As these are very powerful data structures and hold a lot of information about the string upon which they are built on, the find numerous applications such as exact and approximate pattern matching, calculating longest common substrings or subsequences between two strings etc. We concentrated our study on exact pattern matching using these structures.

The goal of this project was to study suffix arrays and trees with a view to using them to build an index over string attributes in Quickstep. In this report, in section 2, we have first described suffix arrays. We have then explained and compared some of the algorithms that we studied for the construction of suffix array. After this, we have explained our implementation and results based on it. In section 3, we have described suffix trees and then similarly compared different algorithms that we have studied and then described our implementation. In the end, we have presented our conclusions based on our study and implementation.

2. Suffix Array

2.1 Introduction to suffix arrays:

Given a string S, a suffix array over S is an array containing lexicographically ordered sequence of all suffixes in S. If S[0,N-1] is a given string, then the suffixes are S[N-1, N-1], S[N-2, N-1]....,S[1, N-1] and S[0, N-1]. A suffix array SA represents all these suffixes in a sorted order.
Example: Consider the word **cactus**.

c	a	c	t	u	s
0	1	2	3	4	5

Fig. 1

As can be seen, it contains the following suffixes: **cactus, actus, ctus, tus, us, s.** A sorted ordering of the above suffixes is **actus, cactus, ctus, s, tus, us**. So the suffix array corresponding to this word will be:

1	0	2	5	3	4

Fig. 2

We have first explained how to search a

pattern given a suffix array such as above and then looked into constructing such an array. Finding a pattern in a string is straightforward and can be done using binary search. Following is the pseudocode for the same:

```
S[0..N-1]: The given string
SA: Suffix array over S
left: The left bound of SA currently in use
right: The right bound of SA currently in use
pattern: pattern to search in SA

START
left = 0;
right = N-1;
len = length of pattern
Do while left <= right:
1. mid = (left+right)/2
2. if S[SA[mid], SA[mid]+len-1] = pattern
   return mid
   if S[SA[mid], N-1] > pattern : right=mid-1
   else left=mid+1

return -1
END
```

For example, assume that we have the above suffix array and are asked to find if the string **actu** exists in the original string. In this case, binary search will first check element at index 2, which corresponds to the suffix **ctus.** As ctus is lexicographically larger than actu, the search moves to left and accesses element at index 0 in the array. It corresponds to the suffix **actus**. As all characters till length of the pattern match, the search returns 1, meaning the pattern actu exists at index 1 in the string, which is the case. This search can easily be extended to find all the indices for a given pattern or to count number of times a pattern occurs in the string. As the process for pattern matching given a suffix array is trivial, we concentrated on studying different algorithms for constructing a suffix array.

2.2 Construction of a suffix array:

We studied various algorithms from [2] for construction of a suffix array.

2.2.1 The Doubling algorithm:

In this method, all the suffixes are sorted in stages, first according to first character, then according to first 2 and then 4 and so on. In general, for comparing any two given strings, we require linear time in worst case as we need to compare each character. But here, if at any stage, we have suffixes sorted according to first 2^i characters, then sorting them according first $2^{(i+1)}$ characters takes constant time as the latter can be broken into two strings of length 2^i, for which we already know the ordering. So if n is the length of the input string, then each stage takes O(NlogN) time and there are O(logN) such stages. So the total time complexity of this algorithm is O(NlogNlogN).

2.2.2 Manber and Myer's algorithm:

This algorithm is quite similar to the doubling algorithm, except that at any given stage, we use the radix sort to sort the suffixes rather than a normal comparison based sorting algorithm. So each stage takes O(N) time in worst case and the total time complexity is O(NlogN).

2.2.3 Baeza Yates - Gonnet - Snider algorithm:

This algorithm is suitable for building suffix array when the input string is large enough, so as to not fit inside the memory. Let N be the size of the string and M the the chunk of string brought into the main memory at each stage. In practice, the algorithm runs in N/m stages where m = lM and l is a positive constant such that l < 1. At each stage, we have a suffix array SAext built by the input read till that stage. This array is stored on the disk as it may not fit in the main memory. In each stage, we read a part of the string in memory and then build the suffix array based on this input SAint. So when stage k starts, SAext is the suffix array based on the input S[0, (k-1)m-1] and SAint is built based on S[(k-1)m, km-1]. SAint is then merged with SAext. This is done in two steps by maintaining a counter array. The worst time complexity of this algorithm is O(N logN). However, in most of the practical cases, it works as well as O(N^2 logN).

The following table shows the comparative results between the above 3 algorithms with respect to time and space:

Algorithm	Space	Time
Doubling (2.2.1)	24N	NlogNlogN
Manber and Myer (2.2.2)	8N	NlogN
Baeza Yates -Gonnet-Snider (2.2.3)	8N	(N^3 logN)/M

Fig. 3

2.3 Implementation and Results:

As an initial goal, we implemented construction of suffix array using the doubling algorithm. We focus on building an array in memory. Although the doubling algorithm is not as efficient as the Maner and Myer's algorithm, building an array is not a very frequent task. So if the database (or a text file in our case) is not updated too frequently, the doubling algorithm can be a decent choice for the array construction. Our code takes as input a text file, builds a suffix array over it and then searches for a given pattern in the file using this array. Following is the pseudocode of our implementation:

```
len = length of suffixes we deal with at any step k (2^(k-1))
suffix_t : structure that holds index, rank, next_rank (rank of the suffix len characters after this suffix)
suffixes : vector of the above structures
prev: vector that contains ranks of suffixes in previous step

START

For step 1, populate suffixes and prev as per length = 1

Do for each step:
1. Update each element of the suffixes vector using ranks corresponding to previous step from the vector prev
2. Sort the vector suffixes
3. Update prev as per the ranks calculated for each suffix from this step.

END
The suffix array is basically the vector suffixes and we only need to consider the attribute index.
```

We referred [4] for implementing suffix array construction. We experimented with various file sizes such as 50KB, 100KB, 200KB, 500KB and 1000KB. The following table shows the results with respect to time required for constructing the array as well as average time for searching a given pattern in this array.

File Size (in KB)	Time for Suffix Array Construction (in ms)	Time For searching a pattern (in micro secs)
50	412.9	10
100	774.167	11
200	1536.69	12
500	3996.82	15
1000	8663.67	16

Fig. 4

Following is a graph that shows the suffix array construction times as we vary the file size:

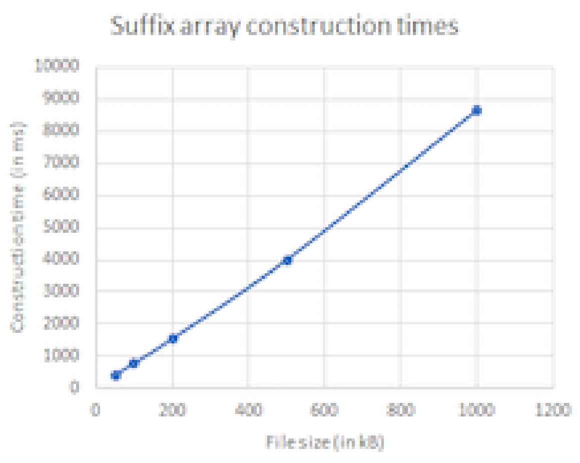

Fig. 5

As expected, array construction time increases as size of the input file increases. The graph shows this increase to be almost linear.

The next graph shows pattern searching times in a suffix array as we vary the file size:

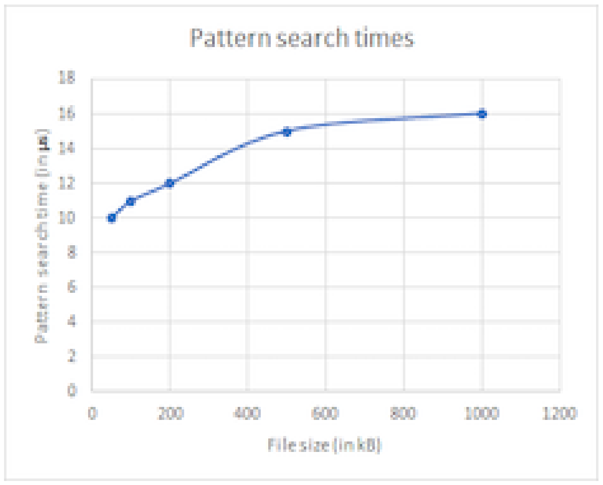

Fig. 6

The above search times are for patterns which do not exist in the text file. The reason was, a meaningful comparison cannot be done with respect to file sizes for existing patterns. It is because different patterns can take different time, some may be found at the start of the search whereas some towards the end. Whereas if we search for patterns that do not exist, the pattern won’t be found after an exhaustive binary search on the array. This also gives an idea of the worst case times. From the above graph, as expected, time increases almost logarithmically as file size increases.

3. Suffix Tree

3.1 Introduction:

Suffix Tree is another way of string matching which supports exact string matching and also supports search using regular expressions. Suffix trees are trie like structures representing all the suffixes of a string. The first step is that, given a string S, build all the suffixes for the given string. In order to get better performance, suffix trees are kept in memory. Taking an example of the word **cactus**, the suffix tree for that would like like the one in Fig. 7.

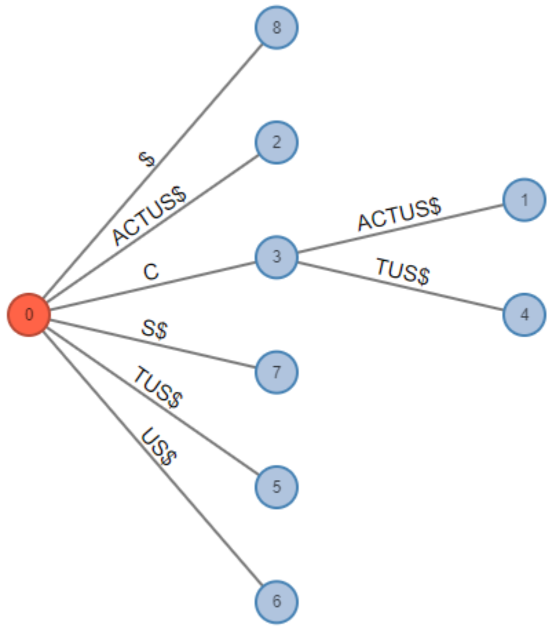

Fig. 7

In the above figure, $ is used as a terminating character. By adding this terminating character to the end of the string we ensure that no suffix is a prefix of another. As we can see, the suffix tree contains all the suffixes which can be formed for the word cactus like, **actus**, **cactus**, **ctus**, **s**, **tus**, and **us**. Suffix trees are useful because they can efficiently answer many questions about a string, such as how many times a given substring occurs within the string, which we can easily get once we have a suffix tree as shown in Fig. 7. With respect to memory requirements, suffix trees take lot of memory and hence this is one of the drawbacks of suffix trees as compared to suffix arrays. The pseudocode for searching a given string in a suffix tree is as shown below:

```
root: Root node of the suffix tree
current: The node in suffix tree where search has reached
pattern: Pattern to be searched in the suffix tree
index: index in pattern till where search has completed


START
current = root
index = 0

Do while pattern is not consumed:
1. if current does not have edge for
        pattern[index] : return false
   else :
   1.1 Read each character c from edge
        if c = pattern[index]:
           increment index
        else : return false
   1.2 Update current

return true
END
```

We referred [5] to implement searching a pattern in a suffix tree. Here, when a string is to be searched in an already constructed suffix tree, we start from the root of the suffix tree and traverse over the given string and check if there is an edge which contains the current character being traversed. If so, we continue, else we return indicating that no such pattern was found in the suffix tree. Let us take an example to understand how the search in the suffix tree works with the above algorithm. Taking the pattern to be searched here as **actus**, initially the search starts at the root of the suffix tree.

starting character of the given pattern. We continue this process of traversing the tree for the rest of the characters like 'c', 't', 'u', 's' till we come to the end of the given pattern. As for this pattern, when it comes to the end we find that the pattern exists in the tree and hence return True.

Let us take another example, this time a pattern that is not present in the tree. Let the pattern to be searched be acy. The initialization steps remain the same for this as explained in the previous example. We now start traversing from the root and check if there is an edge with a character 'a', and we see that we have one and then we continue by picking the next character from the pattern, 'c', we see that even this character 'c' is in the tree. Next when we go to 'y', there is no such character present in the tree so it returns a false indicating that no such pattern exists in the tree.

3.2 Construction of Suffix Tree:

We have studied various algorithms for building a suffix tree from [2], [5], [6], some of them are explained below.

3.2.1 Naive Algorithm:

In the naive algorithm which we have implemented, given a string S which has a length l, we enter a single edge in the suffix tree with the suffix S[1..l] and a $ to indicate the termination of the string into the tree. Continuing this process we then successively enter the suffix S[i..l]$ into the tree. In this way starting from the root, the suffix tree is built.

3.2.2 Ukkonen's Algorithm:

Ukkonen's is a linear time algorithm to build suffix trees. Here, the tree is built by taking the first character of the string, calling it the implicit tree and then adding successive characters of the string until the tree is complete, i.e this algorithm builds implicit suffix tree Ii for each prefix S[1..i] of S, where S is the given string. The time complexity for the algorithm is O(N), and Ukkonen's algorithm builds the suffix tree by iteratively expanding the leaves of partially constructed suffix trees in the previous steps.

3.2.3 McCreight's Algorithm:

A suffix tree of a string S, is a compressed trie of all the suffixes of the sequence S$. In this algorithm, it iteratively builds tries with trie of sequences S$[1..N+1], S$[2..N+1] and so on till S$[i..N+1] where i is between 1 and n+1 and n is the length of the given string. The major idea here is to efficiently insert S$[i..n+1] into the tree so that we do not spend O(N^2) time. This algorithm is also a linear algorithm similar to the Ukkonen's Algorithm.

3.2.4 TDD Technique (PWOTD Algorithm):

Top-Down Disk-based technique uses less amount of memory and this is achieved by buffering.The TDD technique has a suffix tree construction algorithm called the PWOTD (Partition and Write Only Top Down) Algorithm. This algorithm has two phases involved in it, in the first phase, partitions are created for various prefix lengths and then in the second phase wotdeager algorithm is used for building the suffix tree. Considering an example, consider the string **cactus**, taking the prefix length as 1, we would have 5 partition suffixes, one for each character in the string. The suffix partition of the character **c** would be {0,2}, representing the suffixes {**cactus$**, **ctus$**}. As indicated in [1], the algorithm takes O(n^2) time for the construction of suffix trees, but TDD algorithm seems to outperform Ukkonen's algorithm for cached architectures.

These are the four algorithms we have studied and below is a table in Fig. 8 that compares the algorithms with respect to time complexity:

Algorithm	Time Complexity
Naive Algorithm	O(N^2)
McCreight's Algorithm and Ukkonen's Algorithm	O(N)
PWOTD Algorithm	O(N^2), but better for an architecture that has caching.

Fig. 8

3.3 Implementation and Results

We have implemented the naive algorithm and were successfully able to create a suffix tree for a given string and search for a pattern or substring. We then moved on to building suffix trees for strings in a file and we found out that even for 10KB size file, it couldn't be scaled up to generate the suffix tree and we need to use some better algorithm than the naive algorithm. It is also evident from the result that, the memory requirement for suffix trees is high as compared to the memory requirement for suffix arrays as discussed earlier.

Following is the pseudocode for construction of suffix tree:

We referred [6] and [7] for building a suffix tree. In the pseudocode for the construction of suffix tree we see that given a string S, we initially append $ to the end of the string. As discussed earlier this special character is appended to the end of the string to ensure that no suffix is a prefix of another.

```
S[0….N-1]: String over which suffix tree is to be built
T(k): Intermediate suffix tree after any iteration k

START
Append "$" to the given string

In each iteration k, we try to build T(k+1) from T(k)
Do for each suffix starting from 0 to N-1:
1. Start at the root of T(k)
2. Find longest path from the root which
   matches a prefix of S[k+1….N-1]
3. If match ends in the middle of edge:
   3.1 Break the edge and create a new
       Node
   3.2 Spawn 2 new edges from the new
       node, one each for unmatched
       Suffix and unmatched string from
       Previous edge

END
```

T(k) in the algorithm is defined as the intermediate suffix tree after any iteration k. For each suffix in the given string S, we start from the root of the intermediate suffix tree, traverse down the tree till we find the longest path from root that matches a prefix. Once we reach a point where the match ends, we then break the edge and create a new node. From the newly created node, we insert two more edges. One of these edges represents the unmatched string from the previous edge and the other for unmatched suffix.

4. Observations and Conclusions

In our experiment, we faced problems constructing suffix trees in memory even for files of small size like 10KB. We suspect it to be because of the huge memory requirements of the intermediate and final data structures while constructing them.

Leaving aside the memory requirements, although theoretically, suffix trees can be constructed in linear time, the practical results are not so encouraging. Better time complexity need not necessarily mean better performance in practice. For example, the Ukkonen's algorithm suffers from a bad locality of reference as it relies on traversal using the suffix links, which lead to a lot of random references.

We also need to figure out an efficient way for searching patterns with regular expressions. Such patterns will increase the complexity as then there can be multiple paths to search at some points instead of narrowing down to one.

We were able to construct suffix arrays for files as large as 1000 KB and search patterns in them. Although this was done in memory, we can extend this to on disk construction, using an algorithm such as the BaezaYates-Gonnet-Snider algorithm and its variants.

5. References

[1] Practical Suffix Tree Construction
http://www.vldb.org/conf/2004/RS1P3.PDF

[2] A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
https://www.researchgate.net/publication/47841517_A_Theoretical_and_Experimental_Study_on_the_Construction_of_Suffix_Arrays_in_External_Memory

[3] Suffix Cactus:A Cross between Suffix Tree and Suffix Array
https://link.springer.com/content/pdf/10.1007/3-540-60044-2_43.pdf

[4]
https://web.stanford.edu/class/cs97si/suffix-array.pdf

[5]
http://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/

[6]
http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/

[7]
https://www.hackerearth.com/practice/data-structures/advanced-data-structures/suffix-trees/tutorial/