Abstract and Summary of "Efficient Implementation of Suffix Trees"

Vema Teja Krishna, Rohit Kakulpathi, Alla Gopi Karat
Department of Electrical and Electronics, Birla Institute of Technology and Science, Pilani, Rajasthan, India
f2014166@pilani.bits-pilani.ac.in, f2014189@pilani.bits-pilani.ac.in,f2014278@pilani.bits-pilani.ac.in

I. INTRODUCTION

The paper "Efficient Implementation of Suffix Trees" by Andersson and Nilsson starts with the motivation why the efficient implementation of string searching is an important issue, it is very common that we search for a word or a small string in a text or a document and if we go by the brute-force method which traverses the whole document word by word, then it will take a long time and will also have a noticeable effect on computer based on its response time, etc. But if we can compromise a bit by doing a pre-work by using a different data structure and compressing the information, we can save a lot of time in searching.

One of the most commonly used data structure used is a suffix tree. A suffix tree is nothing but a trie in a compressed format containing all the suffixes of the given data with their positions in the data and their values. In general the construction of such a suffix tree for a string or data α takes space and time in linear in the length of α. The suffix trees should only be implemented when while making a search operation a few mistakes are allowed.

A suffix tree for a string α of length λ is a tree such that:

- The tree has exactly λ leaved which are numbered from 1 to λ.
- Every internal node has at least two children except for the root.
- Each edge is labelled with a non-empty substring of α.
- Out of all the edges coming out of a node no two edges can have labels starting with the same character.
- The string is obtained by concatenating all the string labels found on the path from the root to leaf j spells out suffix α [j...λ] for j from 1 to λ.

A sistring is a semi-infinite tree which is a substring of the data or the text which is defined by its position and continuing to the right as far as necessary to make the string unique. When all the references of such sistrings are stored in a trie forms a suffix tree. This tree is further compressed and optimised more by techniques like Path Compression, Level Compression and Data compression to form a space efficient data structure known as Level Compressed Patricia trie. A normal binary trie after path compression gives a Patricia tree and on level compression gives Level Compressed Patricia trie.

Path compression on a normal binary trie, at each node an index is used to indicate the character used for branching that particular node. This additional information is used to remove all the internal nodes which are having an empty subtree and to convert a normal binary trie into a patricia tree. A patricia tree storing 'q' strings has exactly '2q-1' nodes. A patricia tree can be represented in a space efficient manner by storing the nodes in an array. Each node is represented by two numbers one which tells us about the number of bits to be skipped and

which is a pointer to the left child of the node and the pointer to the right child is not required if the children are stored at successive locations in the array as they can be accessed by the next index. Each leaf or the end child will represent a unique sistring and it is also possible that it will contain a pointer to some other data structure.

Level compression is used to reduce the size of the tree, if 'o' highest levels of the data structure are complete that is they don't have any empty children and the 'o+1' level is not complete then we can substitute those 'o' levels with a single node of degree (i.e- number of children) 2^o. This procedure is repeated top-down in the entire tree and we get a structure which is called as LC-trie or the level compressed trie. Based on the experiments performed in the given paper the average height or the depth of a LC-trie is $\Theta(\log^* n)$ but $\Theta(\log n)$ for a normal trie. ($\log^* n$ is the iterated logarithm function: $\log^* 1 = 1$ and $\log^* n = 1+\log^*(\lceil \log n \rceil)$ for n>1).

So, these are two techniques which help in optimising two different aspects but if we combine both of them and use then that is if each internal node of degree 2 that has an empty subtree removed and at each internal node we use a value that indicates the number of bits that have been skipped. On doing this we get the LC-patricia tree. The LC-trie can be very easily implemented by the following method, each node can be represented by three numbers, in which two of them can be used to indicate the number of bits to be skipped and the position of the left most child, and the third can be used to tell the number of children, this will always be a power of 2 and can be represented by using very less number of bits. In general, if a tree has 'y' leaves then$\lceil \log \log(y) \rceil$) bits will be sufficient.

One of the most common application for which it is used is DNA sequences. DNA sequences are generally made of four nucleotide bases which are Adenine-A, Guanine-G, Thymine-T, and Cytosine-C. For example if we consider the first fifteen nucleotide bases of the Epstein-Barr virus:

AGAATTCGTCTTGCT

Figure 1- DNA sequence of Epstein-Barr virus (only first fifteen nucleotides)

If shown in trie it looks like the below, how is it formed is that all the possible unique sistrings are formed and to that corresponding last character the position of the initial character is indicated, for example consider **"CTT"** first character is C then T and then T and the initial character is at position 13 so at the end of path which is at 'T' the index kept is 13. And so on, this procedure is followed all the possible unique sistrings and the indexes are stored.

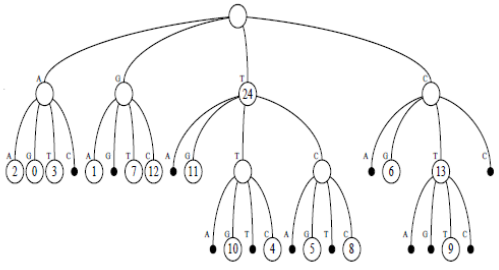

Figure 2- Normal trie representation of the DNA sequence

On conversion to a binary trie it looks like the below, the only difference between a normal trie and binary trie is that each node can have a maximum of 2 children and in case if there is one empty children then in that case only one child is there.

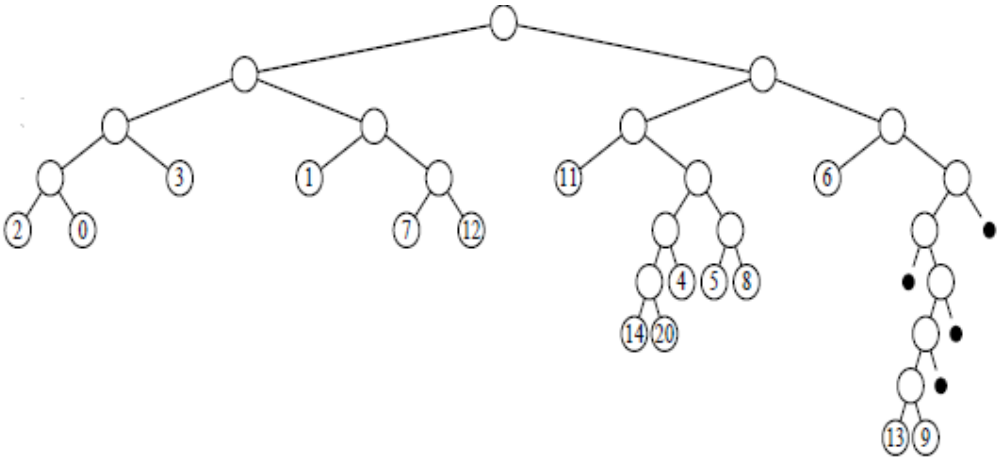

Figure 3- Binary trie for the DNA sequence

And conversion to Patricia tree by using the skip value it looks like the below and additionally only the skip value is added.

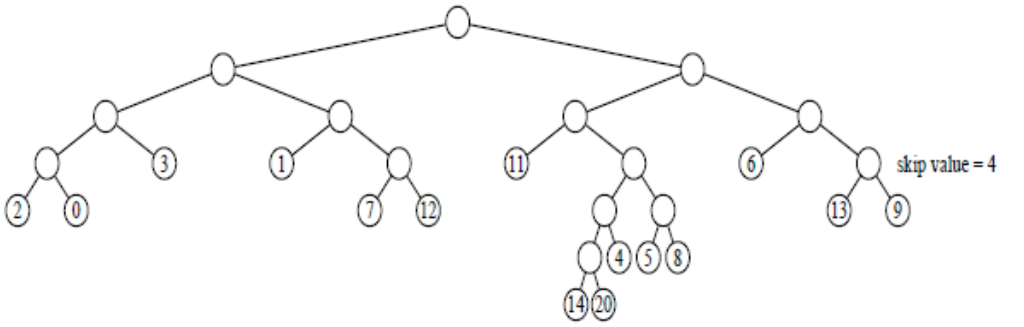

Figure 4- Patricia tree for the DNA sequence

And the corresponding LC-trie is below, here the "u" complete levels are replaced with 2^u children of that node and such procedure on repetition on the whole trie gives the below.

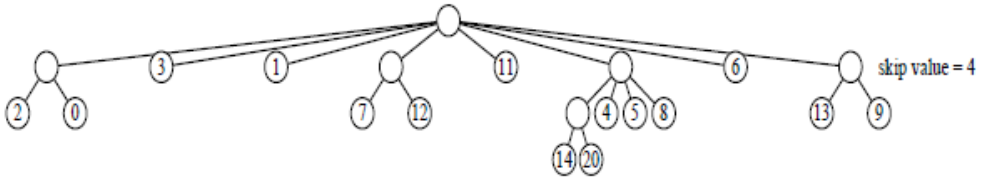

Figure 5- LC-Patricia tree for the DNA sequence

How to find a particular unique sistring? One has to formulate a table as given below so that starting from the root one can find the position of each sistring and as mentioned this data structure is to be only used if we can allow some mistakes as in the case of a node where there is only one children we don't explicitly mention the children and it is possible that the last character is wrong and in our example the last nucleotide can be wrong like in the case og "CTA" and "CTG" so as the possible children for the left node are "A" and "G", so it is difficult

to tell whether it is “CTA” or “CTG”, if the last level has only one children. The table is only formed afte encoding each nucleotide by some number and here we are binary encoding it as 00 for ‘A’, 01 for ‘G’, 10 for ‘T’ and 11 for ‘C’.

S No	Branch	Skip Value	Pointer to the first character
0	3	0	1
1	1	0	9
2	0	0	3
3	0	0	1
4	1	0	11
5	0	0	11
6	2	0	13
7	0	0	6
8	1	4	19
9	0	0	2
10	0	0	0
11	0	0	7
12	0	0	12
13	1	0	17
14	0	0	4
15	0	0	5
16	0	0	8
17	0	0	14
18	0	0	10
19	0	0	13
20	0	0	9

Table 1- Array implementation of the LC-trie above

It is also common that while forming tries like this for any application let it be DNA sequence or a document, usually all the symbols or characters are binary encoded and then formed as an array as above so it often common that all the possible characters are not used for example if we are using 8bit encoding for each character then it is very rare that in a document all the 256 characters will be used so we use a different scheme known as Huffman Coding. In this type of coding the sistring or the symbol which is most commonly used is coded with the least number of bits so that the space complexity is as less as possible and this is one of the most well-known and efficient compression technique.

II. CRITIQUE AND ANALYSIS

A suffix tree can be implemented in the form of a LC-Patricia tree in O(k* logx) time and in O(k) space, where k is the size of the text and x is the size of the alphabet. This can be formed by forming ordinary suffix tree and it is constructed in two steps-

1. Initially, each node of the tree is substituted by a Patricia tree with two children that is a binary patricia tree and the skip values are added. This step takes usually

O (k) time and since the number of nodes usually formed are also in O (k) it is trivial.

2. After the above step, the technique of level compression is applied in a top-down form and as explained above the complete ‘y levels are replaced with a node with 2^y children and it is applied recursively and again as the number of children is of O(k) it takes O(k) time.

In general, there are more simpler methods available where we first construct an array which contains the pointers to all the sistrings of the document or the data and then we sort this array using any standard sorting algorithm which takes O (k logk) time such as Forward Radix Sort,etc. But such cases have a worst case scenario of quadratic order that is of O (k^2), so for some particular types of data this can be efficient. Also, when one is working with large texts then it is difficult to fit them in main memory which only consists of some registers and it has to be stored in secondary memory and access to secondary memory takes much more time than that of main memory. So, in such cases we construct a new data structure such as partial LC-trie, in which there is a cutoff value‘t’ constant, and whenever we get a node which covers less than ‘t’ strings we add a pointer to the suffix tree and the value is called as the cutoff value.

Other such implementations include Trie Implementations, Array Implementations, etc. In Trie implementation we form a k-ary trie where k is the size of the alphabet and each node contains the pointers to the next descendants and all the non-empty pointers are stored in a structure like a linked list to be continuous and are also stored in a balanced binary tree. But this has a large space overhead and is not useful in practical applications.

S No	Data Structure Used	Time Complexity
1	Patricia LC-trie	O (k logn)
2	Trie Implementation	O (k logn) but worst case O (k^2)
3	Normal Traversal	O (k*n^2)

Table 2- Time complexity for finding a sistring for different data structures [1, 3], here k is the size of the alphabet and n are the number of sistrings

Input		Suffix Array		Trie (list)		Trie (array)		Patricia Tree		LC-trie	
File	Size	Steps	Size	Depth	Size	Depth	Size	Depth	Size	Depth	Size
Random text	2	10.0	16					12.4	24	5.0	20
	20	13.3	160					15.6	240	4.6	202
	200	16.6	1600					18.9	2400	4.7	2018
DNA	1.7	9.8	14	15.8	41	6.3	20	12.4	21	5.1	17
	17	13	138	19.1	423	8.2	213	15.9	207	5.6	180
	172	16.4	1378	24.5	4279	11.0	2209	20.3	2067	6.8	1824
FAQ, ASCII	1.9	9.9	15	60.2	42	3.9	328	15.5	23	11.2	22
	19	13.2	155	80.7	431	5.7	3593	21.6	232	15.9	222
	193	16.6	1545	115.8	4313	8.2	36122	30.2	2318	21.6	2207
FAQ, Huffman	1.9	9.9	15					13.3	23	7.2	21
	19	13.2	155					18.3	232	9.9	219
	193	16.6	1545					24.5	2318	13.1	2196

Figure 5- Results for suffix array and suffix trees for some cases, sizes are measured in kB, taken from the given reference paper [1]

Input		Bucket Array			LC-trie		
File	Size	Accesses Aver.	Accesses Worst	Size	Accesses Aver.	Accesses Worst	Size
Random text	200	2.5	5	131	2.9	5	140
DNA	172	2.9	7	131	3.1	6	137
FAQ, ASCII	193	8.2	14	131	4.8	7	64
FAQ, 96-digit		6.8	14	193			
FAQ, Huffman		5.2	14	131	4.7	7	56
bib, ASCII	111	7.7	13	131	4.9	7	34
bib, 96-digit		6.8	13	111			
bib, Huffman		5.1	11	131	4.9	7	30
paper1, ASCII	53	7.0	11	66	4.0	6	31
paper1, 96-digit		5.9	11	53			
paper1, Huffman		4.1	9	66	3.9	6	27
paper2, ASCII	82	7.7	13	131	4.0	6	50
paper2, 96-digit		6.8	13	82			
paper2, Huffman		4.2	10	131	3.9	6	42
progc, ASCII	40	6.9	11	33	4.1	6	22
progc, 96-digit		5.6	11	40			
progc, Huffman		4.1	8	33	4.0	6	20
progl, ASCII	72	7.9	12	66	4.1	6	41
progl, 96-digit		6.9	12	72			
progl, Huffman		5.1	12	66	4.0	6	39
progp, ASCII	49	7.6	13	33	4.1	6	28
progp, 96-digit		6.4	13	49			
progp, Huffman		5.2	12	33	4.0	6	27
trans, ASCII	94	7.1	12	131	4.0	6	61
trans, 96-digit		6.4	13	94			
trans, Huffman		5.1	11	131	4.0	6	57

Figure 6- Results for suffix array and suffix trees for some cases using partial data structures, sizes are measured in kB, taken from the given reference paper [1]

From the results founded from various papers and some from trivial understanding for brute-force methods, it is clear that the LC-Patricia tree which is implemented by the Suffix Tree takes the least time and the least time in comparison to most of the other data structures even in the case of partial data structures where the secondary memory is also used. Only in few applications like DNA sequences the data structures implemented by the Suffix Array performs better but for most of the other applications.

III. REFERENCES

[1] Efficient Implementation of Suffix Trees, Arne Andersson, Stefan Nilsson, Dept of CS, Lund University,Sweden

[2] DA Huffman, A method for the contruction of minimum redundancy codes, In Proc.IRE volume 40

[3] Text Compression, TC Bell, JG Cleary, Prentice Hall, Englewood Cliffs, 1990

[4] Suffix trees, https://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/suffix.htm

[5] A new efficient radix sort, S Nilsson, Annual IEEE Sympos 1994

[6] Path Compression, https://courses.cs.washington.edu/courses/cse326/00wi/handouts/sld035.htm