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](mailto:f2014166@pilani.bits-pilani.ac.in), f2014189@pilani.bits-pilani.ac.in,f2014278@pilani.bits-pilani.ac.in

# 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.

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. 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:



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.

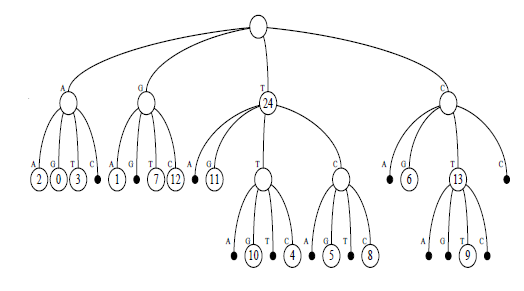


Figure 2- Normal trie representation of the DNA sequence

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.

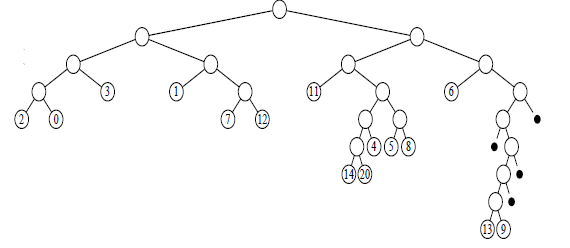


Figure 3- Binary trie for the DNA sequence

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

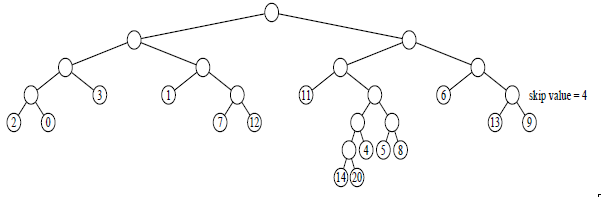


Figure 4- Patricia tree for the DNA sequence

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

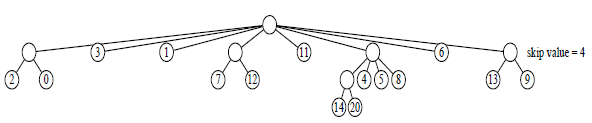


Figure 5- LC-Patricia tree for the DNA sequence

|  |  |  |  |
| --- | --- | --- | --- |
| 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.

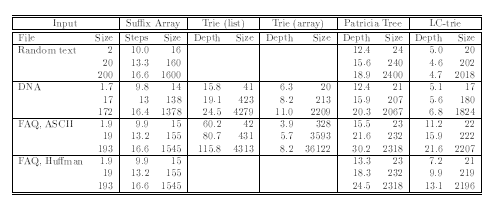


Figure 5- Results for suffix array and suffix trees for some cases, sizes are measured in kB, taken from the given reference paper [1]

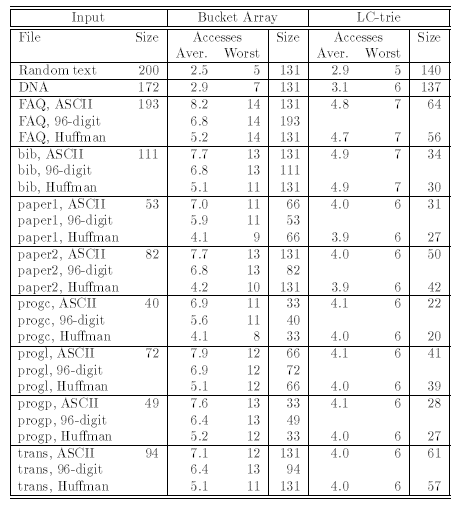


Figure 6- Results for suffix array and suffix trees for some cases using partial data structures, sizes are measured in kB, taken from the given reference paper [1]