Asian Journal of Engineering and Applied Technology
ISSN 2249-068X Vol. 2 No. 1, 2013, pp.7-11

Indexing Genome with the External Construction of Compressed Suffix Tree Using LCP Array

Vijay Kumar Vishwakarma[1] and Abhishek Srivastava[2]
Department of Computer Science and Engineering, Jaypee University of Engineering and Technology, Guna, Madhya Pradesh - 473 226, India
Email: vijaykrvishwakarma@gmail.com, abhishek.srivastava@jiet.ac.in
(Received on 10 April 2013 and accepted on 15 May 2013)

Abstract – **We are proposing the genome indexing algorithm, which depends upon compressed form of suffix trees, in which every node has four parts; suffix array number, suffix start number, LCP count, and a pointer to another node. The proposed algorithm does not use the whole suffix array, it just takes some necessary information like LCP of two suffix array, compare them and suffix start number, to align the suffix to proper position and suffix array number to distinguish among all the partitions. The use of compressed suffix array minimizes the number of trees, eventually; it also minimizes the random access to input data, as it creates the compressed suffix tree for two suffix arrays using pairwise sorting, sequentially.**

Keywords: **Genome Indexing, Compressed Suffix Tree, Data Structure, DNA Indexing**

I. Introduction

Genome word came from the words "Gene" and "Chromosome". It contains the hereditary information of an organism. A genome is an organism's complete set of DNA, including all of its genes. Each genome contains all of the information needed to build and maintain that organism. There are 4 nucleotides in a Genome Sequence; Adenine (A), Cytosine (C), Guanine (G), and Thymine (T).

All four nucleotides or DNA symbol are arranged in a unique manner for 1000 symbols. We will use this property to differentiate the suffix trees and suffix arrays of different partitions.

Genome indexing is a technique used to access the DNA string or Genome sequence and extract that hereditary information. An index is a data structure methodology that improves the speed of data retrieval operations at the cost of slower writes and increased storage space. Indexing can be created using suffix tree data structure, provides the basis for both rapid random lookups and efficient access of ordered records.

All hierarchical data structure includes indexing technology that enables sub-linear time lookup to improve performance, as linear search is inefficient for large datasets. Indexing very large datasets is a tedious task, actually done by automated systems. It is multi-level process, like the cleaning of genomic sequence, partitioning of input datasets, which is larger than main memory, and organizing the data in a data structure.

Suffix tree is a well suited data structure, which can index the genome, efficiently. It builds the tree in linear time and searches the string in linear time. The existing methods like [1]Trellis [1] and [2]DiGeST [2] can index the genomic data up to 3GB. We need a scalable suffix tree algorithm that index the genome further 3 GB.

II. Background

In computer science, a suffix tree (also called PAT tree or, in an earlier form, position tree) [3] is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations.

The suffix tree for a string S is a tree whose edges are labeled with strings, such that each suffix of the S corresponds to exactly one path from the tree's root to a leaf. It is thus a radix tree (more specifically, a Patricia tree) [4] for the suffixes of S. The suffix tree for the string S of length n is defined as a tree such that:

1. The paths from the root to the leaves have a one- to-one relationship with the suffixes of S.
2. Edges spell non-empty strings.
3. All internal nodes (except perhaps the root) have at least two children.

Since such a tree does not exist for all strings, S is padded with a terminal symbol not seen in the string (usually denoted $). This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of S. Since all internal non-root nodes are branching, there can be at most n − 1 such nodes, and n + (n − 1) + 1 = 2n nodes in total (n leaves, n − 1 internal nodes, 1 root).

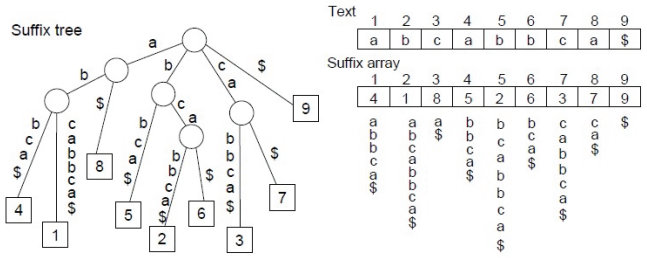

Fig. 1 Suffix tree and suffix array

A. Compressed Suffix Tree

Compressed suffix trees [5] can be implemented in O(n) bits by using compressed suffix arrays and the techniques for compact representation of Patricia tries. The compressed suffix tree occupies space proportional to the text size, i.e. O(n log | Σ |) bits, and supports all typical suffix tree operations with at most log N factor slowdown.

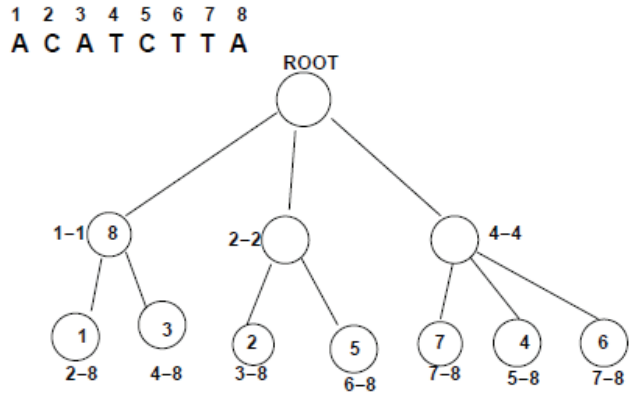

Fig. 2 Compressed Suffix tree

B. LCP Array

The LCP-array [6] stores the lengths of the longest common prefixes of lexicographically adjacent suffixes, and it can be computed in linear time. We have modified the LCP array according to our algorithm and store some additional information; suffix start number with LCP values of respective suffixes. Suffix array with LCP array shown in figure 3.

III. Problem Definition

Given a string X = X1, X2…XN-1 to be a sequence of N symbols. The first N – 1 symbols are over a finite alphabet Σ, Xi Ɛ Σ ($0 \le i < N - 1$). The last symbol XN-1 is unique and not in Σ (called as string terminals).

Given a genome sequence of length N, we have to minimize the input output performance by reducing the merging time of suffix trees.

IV. Proposed Method

The proposed algorithm works in three steps:

i. Input Preprocessing

In this step, we encode the input string and make partition, accordingly: A-00, C-01, G-10, T-11.

For example, human genome of 3GB, by encoding we can compress the data to $(3*2^{30})/4$ = 768 MB, which can now process in main memory. The input string X of size N into k partitions, such that k=2r, where r=N/M, r should be at least 2. For partitioning, we are using existing Larsson's algorithm, which uses quick sort with partition strategy. Nesper Larsson [7] develops this algorithm for the partitioning and sorting according to lexicographical order [8].

ii. In-memory sorting of suffixes

In this step we generate suffix arrays for each pair of partitions. We compute the LCP between two suffixes. Then, sort them according to lexicographical order. A lexicographical order is the alphabetical order as in a dictionary. We put LCP value and starting index of suffix in the suffix array, which is to be used in merging those suffix arrays as compressed suffix tree (CST). We use LCP information for pairwise sorting [9] of two suffix arrays. For in-memory sorting we use Larsson's quicksort algorithm which divides and sorting lexicographically.

string	a	b	a	b	a	a	b	α
index	1	2	3	4	5	6	7	8
Suffix start	5	6	3	1	4	2	7	8
	a	a	a	a	b	b	b	α
	a	b	b	b	a	a	α	
	b	α	a	a	a	b		
	α		a	b	b	a		
			b	a	α	a		
			α	a		b		
				b		α		
				α				
LCP	0	1	2	3	0	2	1	0

a	a	b	a	a	β
1	2	3	4	5	6
1	4	2	5	3	6
a	a	a	a	b	β
a	a	b	β	a	
b	β	a		a	
a		a		β	
a		β			
β					
0	2	1	1	0	0

Fig. 3 Suffix arrays and LCP array

iii. Pairwise sorting and merging

At the end of sorting step, we have on disk k suffix arrays for k partitions (of total size N). Then we have to create a compressed suffix tree for each pair of suffix arrays, by comparing their LCP value. Let there be two suffix arrays; A and B. If LCP of A is less than or equal to LCP of B, then put the regarding suffix into the output buffer. Continue the process for all pairs of suffix arrays. There will be k-1compressed suffix tree for k suffix arrays. We use 2pmms algorithm [10] to merge all the suffix arrays. Note the lexicographical order must be maintained.

Structure of node for compressed suffix tree

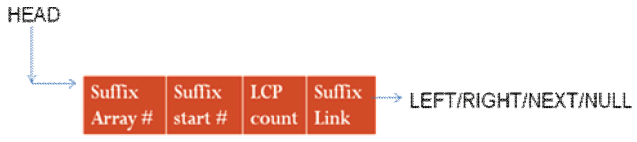

Fig. 4 Node of Compressed suffix tree

Create k number of input buffers for k number of partitions and use the remaining amount of main memory as output buffer. Using two phase multi-way merge sort for external memory, then read input block from two suffix arrays (LCP values) and compare them, if LCP of SAx is smaller or equal to SAy then, we write SAN(suffix array number), SSN(suffix start number), and LCP (longest common prefix) value to the output buffer as a node. If the output buffer is full then, we read the output buffer and write all the nodes to the secondary memory in a file, where all nodes of the compressed suffix tree exists.

Using this approach we reduced the number of suffix trees and create the compressed suffix tree in sequential order, so that searching will takes place in sequential order. There is no random access to the input string. So, we can say that 100 percent of random access is removed.

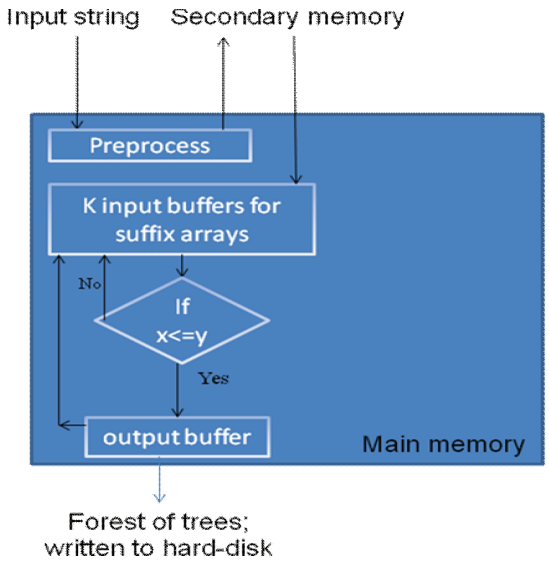

Fig. 5 Flow of the proposed algorithm

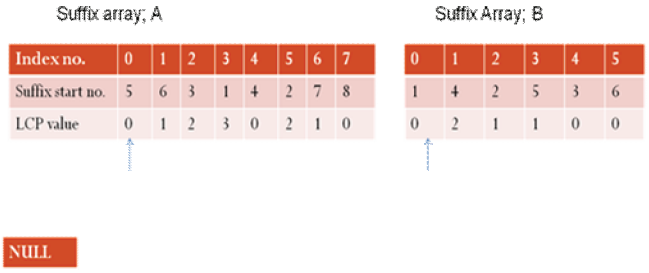

Fig. 6 Initial empty compressed suffix tree

Above Figure 6 shows the empty suffix tree, we have to compare the LCP(A[0]) and LCP(B[0]), LCP(A[0]) is equal to LCP(B[0]), then insert the NODE regarding LCP(A[0]) with relevant information like suffix array number, suffix start number, LCP value of corresponding suffix. Below Figure 7 shows the suffix tree after insertion of NODE(A,5,0). The NODE(A,5,0) is linked to the root of the suffix tree, which was NULL (in previous Figure 6), after the insertion of NODE(A,5,0) the pointer of suffix array A will be incremented and now compare LCP(A[1]) and LCP(B[0]).

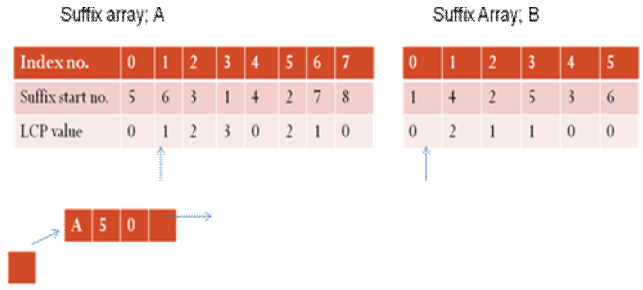

Fig. 7 Insertion of DE(A,5,0)

Fig. 8 Final compressed suffix tree after inserting all nodes

Similarly, compare the LCPs values of LCP array [6] A and B, and whichever is smaller or equal than insert the node in suffix tree, rewardingly, give priority to former suffix array (here suffix array A) if the LCP values of both the suffix arrays are equal. After inserting all nodes into the suffix tree, will be shown in Figure 8. At the end of merging the nodes of both the suffix array, we check the output buffer is full or not. If full, then, we write the nodes of output buffer to secondary memory, otherwise continue the merging process with next suffix arrays. (E.g. BC, then CD and so on).

Similarly, create the compressed suffix tree for BC, CD and so on, sequentially. We have collection of nodes in the output buffer, if the output buffer is full, then we will empty it to secondary memory by writing all the nodes. In this way, we have all the nodes of the compressed suffix tree in a file, which has information about all the connected nodes. All files are linked with each other by the tail (t<1000), the tail is the prefix of next partition, which is attached to the previous partition for differentiating the partition and its suffix arrays.

V. Experimental Results

The simulation has been performed on Ubuntu Linux 10.4, with 3 GB RAM, 4MB L2 cache, Intel i3 core processor of 2.26 GHz. Developed in c++ (gcc compiler) and executed in TPIE environment.

Table I Running Time of Different Algorithm

Algorithm/ Dataset	250 MB	500 MB	3000 MB
	Running time in minutes		
Trellis	107	202	1260
DiGeST	71	126	780
Proposed	9	28	244

The reason of running time of proposed algorithm is; first, the input data is encoded and compressed, and hence, can process more data in main memory. Compressed data lead to less number of partitions and less number of suffix arrays, by which LCP array is created with useful and relevant information. Second, the input data are accessed, sequentially and while merging two suffix arrays as one compressed suffix tree is also in sequential order. Thus, there is no random access to the input data. Finally, creating compressed suffix tree is an advantage of running time.

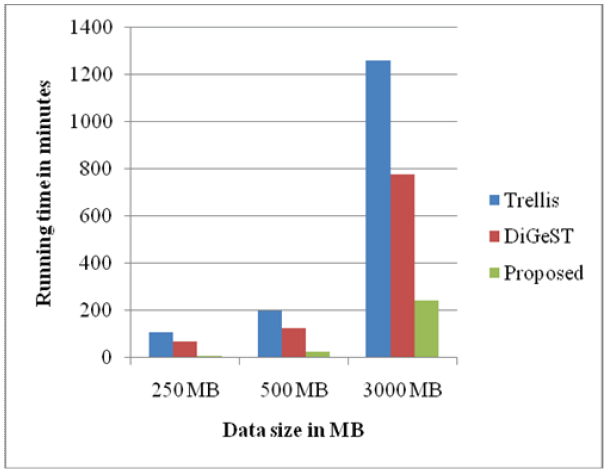

Fig. 9 Comparison among running time of algorithms (Bar chart)

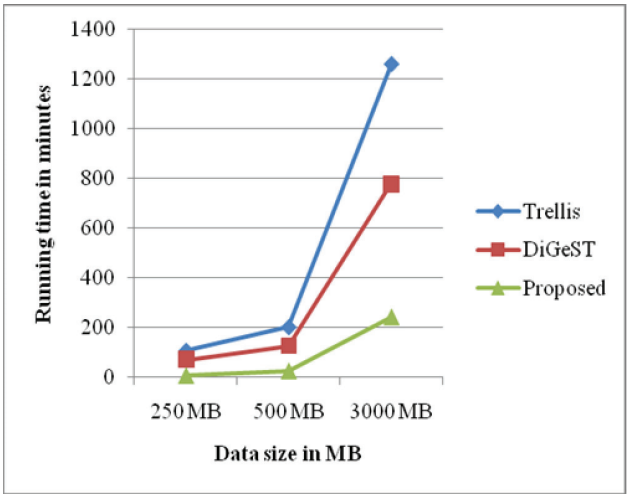

Fig. 10 Comparison among running time of algorithms (Line chart

By seeing the results above, we can say that the proposed algorithm is much better than that of Trellis and DiGeST algorithms, in terms of time complexity, space complexity and I/O complexity.

The advantage of using a compressed suffix tree is that we can save secondary memory space for the number of generating trees, and one more advantage is that the use of suffix link, efficiently, which makes all the nodes and compressed suffix tree connected and hence, the search time of any gene or DNA word, will be easier and faster. The use of suffix link with a compressed suffix tree is efficiently minimized the random access of input data. The whole input data is accessed sequentially.

VI. Conclusion

The proposed algorithm is better in terms of time complexity and it can scale itself to index genome further 12GB, but DiGeST algorithm is limited to scale the data up to 12GB. So, we can say that proposed algorithm is scalable because the algorithm performs in LCP array construction.

The algorithms perform well in practice and can be successfully used for indexing all substrings in databases of long strings, especially of sequenced genomes. We believe that these algorithms are important steps towards a fully scalable solution for constructing full-text indexes on disk for inputs of any type and size. Once this is done, a whole world of new possibilities will be opened, especially in the field of biological sequence analysis.

References

[1] Benjarath Phoophakdee and Mohammed J. Zaki, "Genome-scale disk-based suffix tree indexing". SIGMOD '07: Proceedings of the ACM SIGMOD *International Conference on Management of Data.* ACM. pp. 833–844, 2007.

[2] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton . "A new method for indexing genomes using on-disk suffix trees". CIKM '08: *Proceedings of the 17th ACM Conference on Information and Knowledge Management.* ACM. pp. 649–658, 2008.

[3] http://en.wikipedia.org/wiki/Suffix_tree

[4] Donald R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM,* Vol. 15, NO 4, pp. 514-534, 1968.

[5] Niko Välimäki, Wolfgang Gerlach, Kashyap Dixit and Veli Mäkinen, "Compressed Suffix Tree - A Basis for Genome-scale Sequence Analysis". Bioinformatics, 23(5), Application note, pp 629-630, 2007

[6] Simon Gog, Enno Ohlebusch, "Fast and Lightweight LCP-Array Construction Algorithms", ALENEX, 2011.

[7] N.J. Larsson and K. Sadakane, "Faster suffix sorting", Tech. Rep. LUCS-TR: 99-214 of the Dept. of Comp. Sc., Lund University, Sweden, 1999.

[8] http://en.wikipedia.org/wiki/Lexicographical_order

[9] Md. Jahangir Alam, Muhammad Monsur Uddin, Mohammad Shabbir Hasan, Abdullah Al Mahmood, " Pair Wise Sorting: A New Way of Sorting", *International Journal of Computer Science and Information Security,* Volume 8:9, pp. 116-120,2010.

[10] Simonas Salteni, "External memory sorting", Deptt. Of Computer Science, Aalborg University, Denmark, LNCS, pp 1-7, 2001.