LETTER

A Storage-Efficient Suffix Tree Construction Algorithm for Human Genome Sequences**

Woong-Kee LOH† *and* **Heejune AHN**†††*a)**, Members***

SUMMARY The suffix tree is one of most widely adopted indexes in the application of genome sequence alignment. Although it supports very fast alignment, it has a couple of shortcomings, such as a very long construction time and a very large volume size. Loh et al. [7] proposed a suffix tree construction algorithm with dramatically improved performance; however, the size still remains as a challenging problem. We propose an algorithm by extending the one by Loh et al. to reduce the suffix tree size. As a result of our experiments, our algorithm constructed a suffix tree of approximately 60% of the size within almost the same time period.

key words: *storage-efficient suffix tree, human genome sequences, divide-and-conquer*

1. Introduction

Since human DNA sequences were announced as the product of the Human Genome Project (HGP), a lot of research activities are under way for practical application of the sequences. One of them is aligning short genome subsequences of 100 ∼ 1000 length to the human genome sequences [3]–[5]. This application is getting more attention along with the recent advances in DNA sequencing technology***. For fast alignment, it is essential to use efficient indexes; the suffix tree, suffix array, and compressed suffix array are the most widely adopted [1]–[5].

The suffix tree supports very fast alignment and can also be used in many other applications such as finding frequent subsequences, common subsequences, and maximal palindromes [3], [8], [9]. However, it has a couple of shortcomings; it requires a very long time to construct and also occupies a large volume size [6], [8]. Loh et al. [7] proposed a suffix tree construction algorithm to cope with the first shortcoming. The algorithm dramatically reduced the construction time by making the most of recent multi-core CPUs and minimizing disk access costs.

In this paper, we propose a suffix tree construction algorithm, which is an extension of the one by Loh et al. [7], for coping with the second shortcoming of the suffix tree.

Manuscript received July 26, 2011.

†The author is with the Department of Multimedia, Sungkyul University, Korea.

††The author is with the Department of Control & Instrumentation Engineering, Seoul National University of Science and Technology, Korea.

*Corresponding author

**This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant number: 2010-0025001).

a) E-mail: heejune@seoultech.ac.jp

DOI: 10.1587/transinf.E94.D.2557

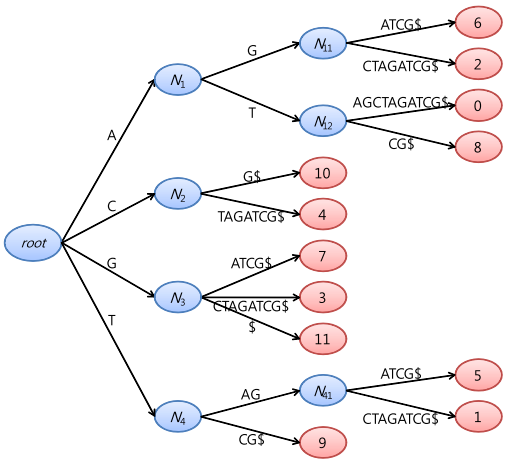


Fig. 1 Suffix tree for a sequence X = ATAGCTAGATCG$ [7].

Since it is impossible to reduce the number of nodes in the suffix tree for a given sequence, our approach reduces the storage size of the suffix tree nodes. As a result of our experiments, compared with the one by Loh et al. [7], our algorithm constructed a suffix tree of approximately 60% of the size within almost the same time period.

Figure 1 shows the suffix tree for a genome sequence X = ATAGCTAGATCG$ [7]. The symbol '$' is attached at the end of X to prevent any suffix from being the prefix of any other suffixes and hence to enable the efficient processing of suffixes. A genome sequence $X = x_0x_1 \ldots x_{n-1}\$$ of length n contains n suffixes $S_i = x_i \ldots x_{n-1}\$$ $(0 \leq i < n)$. The suffix tree for X should have n terminal nodes; they have one-to-one correspondence with the suffixes and contain the positions of the corresponding suffixes. For instance, by searching for a suffix S = AGATCG$ using the suffix tree in Fig. 1, we go through internal nodes N_1 and N_{11} and then reach the terminal node 6, which indicates that the suffix S is located at position 6 in X.

2. Related Work

In this section, we first briefly introduce the suffix tree construction algorithm by Loh et al. [7], which we call the *FAST* algorithm in this paper. FAST is based on a 'divide-and-conquer' strategy and divides all the suffixes in the genome sequences into multiple partitions. For a pre-specified prefix

***DNA sequencing is determining the sequence of chemical base pairs in a DNA molecule.

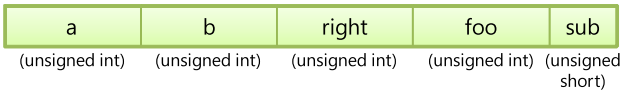


Fig. 2 Node data structure of the FAST algorithm: Information on the inbound edge is contained together [7].

length p, each partition is composed of the suffixes with the same prefix of length p, and hence there are 4^p partitions.

FAST constructs the suffix subtree for each partition. Since the suffixes with different prefixes cannot be inserted in the same suffix subtree, the construction of a suffix subtree for each partition can be performed separately from the others. FAST assigns the construction of different suffix subtrees to different processes. The recent design trend of CPUs not only raises their clock speeds but also builds multiple independent cores in a single CPU to enable intra-CPU parallel processing. By using a multi-core CPU, the processes launched by FAST are executed in parallel by different cores, and therefore the performance of suffix tree construction is significantly improved.

Moreover, FAST almost eliminates random disk accesses, which were the major cause of performance degradation of the previous disk-based suffix tree construction algorithms [1], [3], [8], [9]. In general, even when accessing the same amount of disk volume, the performance greatly depends on disk access patterns; sequential access can improve the performance more than hundred times compared to random access. FAST constructs the suffix subtree for each partition in a contiguous memory chunk and then stores the chunk sequentially in disk[†]. This helps to minimize the disk access costs and hence the performance is dramatically improved.

Figure 2 shows the node data structure of the suffix tree constructed by FAST. Fields a and b ($a \leq b$) indicate the start and end positions of the inbound edge label. For instance, the inbound edge label of terminal node 6 in Fig. 1 is found at position (8, 12) in X, and hence the field values are $a = 8$ and $b = 12$. The field *right* stores the pointer to the next sibling node; the pointer is not a physical memory address but a relative offset from the start of the memory chunk in which the suffix subtree is constructed. By using such pointers, even when the suffix subtree is saved on disk and then reloaded into a different position in memory, there is no need to adjust the pointers. The field *foo* indicates a pointer to the leftmost child node in an internal node or the position of the corresponding suffix in a terminal node. The field *sub* indicates the number of the chromosome referenced for processing the node. The human genome sequences are composed of 22 chromosome pairs (numbered $1 \sim 22$) and x/y (sex) chromosomes (46 in total). Chromosome 1, which is the largest, spans about 247 million base pairs.

Hunt et al. [3] proposed the first algorithm to construct the suffix tree for human genome sequences. Since then, a few more algorithms with improved performance were proposed [1], [8], [9]. Since the suffix tree for human genome sequences has a very large volume size, these algorithms construct disk-based suffix trees. The common shortcomings of these algorithms are that they cannot fully utilize the recent multi-core CPUs and incur random disk accesses. FAST tackled these shortcomings and achieved a dramatic performance improvement; however, there was no improvement in the storage size of the suffix tree by FAST.

3. Suffix Tree Construction Algorithm

Our algorithm proposed in this paper is an extension of FAST [7] and constructs the suffix subtree of a much smaller size than FAST. In general, given a set of suffixes, any different algorithms should construct the same suffix subtree regardless of the order of suffix insertion. That is, it is impossible to reduce the number of nodes in the suffix subtree by any means, and therefore our algorithm reduces the storage size of the nodes.

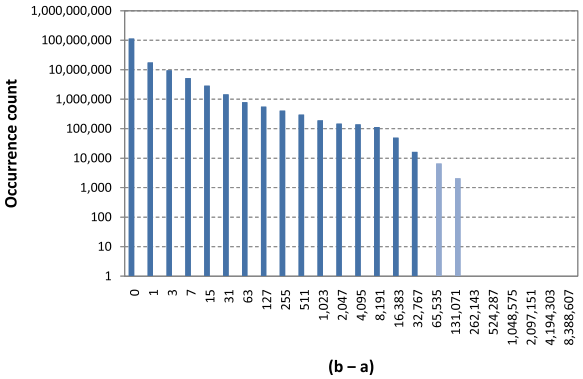
The methods to reduce the node size in our algorithm are (I1) using the correlation between the field values and (I2) concatenating the chromosomes to form a single, long sequence. The method (I1) is based on the following observations between the field values in Fig. 2:

- $O1$ In most of the internal nodes, the field difference ($b - a$) (≥ 0), i.e., the length of the inbound edge label is not larger than 32767 ($= 2^{15} - 1$).
- $O2$ In most of the terminal nodes, the field difference ($a - foo$) (≥ 0) is not larger than 32767.
- $O3$ In terminal nodes, the field b is always n. Hence, the length of the inbound edge label is readily computable without storing the field b.

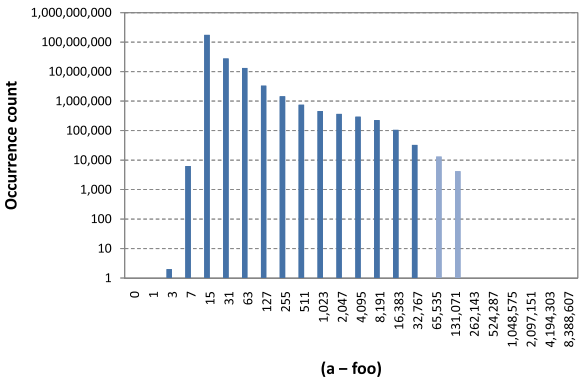
The fields a, b, and *foo* can have values in the range of $0 \sim 4{,}294{,}967{,}295$ ($= 2^{32} - 1$).

Figure 3 is plotted using chromosome 1 and justifies the observations above. Figure 3 (a) justifies the observation $O1$ and shows the occurrence count of every possible ($b - a$) value. Note that both of horizontal and vertical axes are represented in the log scale. Each horizontal value x_i ($1 \leq x_i \leq 8{,}388{,}607$) in the figure actually represents a range of values $x_{i-1} \sim x_i$. For instance, the occurrence count for 255 actually represents the sum of occurrence counts for every ($b - a$) value in the range of $128 \sim 255$. As shown in the figure, approximately 99.98% of the internal nodes satisfies the observation $O1$. Meanwhile, there exist a number of terminal nodes such that the lengths of their inbound edge labels are larger than 32767. Figure 3 (b) justifies the observation $O2$ and shows the occurrence count of every possible ($a - foo$) value. Approximately 99.97% of the terminal nodes satisfies $O2$. The observation $O2$ indicates that the concatenation $L = L_1 \oplus \cdots \oplus L_{t-1}$ of the inbound edge labels L_i ($1 \leq i < t$) of ($t + 1$) nodes $N_0, \ldots, N_t$ ($N_0 = root, N_t = T$) on the path from the root to a terminal node T has the length that is not larger than 32767, i.e., $Len(L) \leq 32767$.

[†]The size of the suffix tree can be adjusted by changing p in accordance with the size of the available memory.



(a) Observation $O1$: the field difference $(b - a)$ is not larger than 32767 in most of the internal nodes.

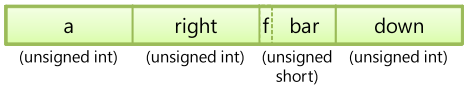


(b) Observation $O2$: the field difference $(a - foo)$ is not larger than 32767 in most of the terminal nodes.

Fig. 3 Observations on the node fields.

The method (I2) in our algorithm is to concatenate the chromosomes to form a single, long sequence[†]. The method (I2) brings two advantages as follows. First, there is no need to deal with duplicate suffixes. When each chromosome is processed separately, there exist the duplicate suffixes from different chromosomes. It is clear that the complexity of an algorithm and a data structure increases to deal with such duplicate suffixes. Second, the information needed to be stored in a node can be covered by using only the fields a, b, and foo; there is no need to use the field sub in Fig. 2. It is because the concatenation of the human genome sequences has the size of 3 Gbp ($< 2^{32}$) and the four-byte (32-bit) unsigned int type fields are sufficient to represent any positions therein. A shortcoming of this method is that it can incur false positive errors in the result of alignment. However, these errors can be removed in the post-processing step by maintaining the size of chromosomes.

Figure 4 shows the node data structure defined in our algorithm based on the methods (I1) and (I2). The fields a and $right$ common in both nodes are used in the same manner as in Fig. 2. The most significant bit f in the field bar is the flag distinguishing the internal and the terminal nodes. The remaining 15 bits in bar, which can represent a value in the range of $0 \sim 32767$, stores $(b - a)$ in an internal node or $(a - foo)$ in a terminal node. The field $down$ in the internal node indicates a pointer to the leftmost child node. While



(a) Internal node.

(b) Terminal node.

Fig. 4 Data structure of our algorithm: It has a smaller size than the FAST algorithm.

both the internal and the terminal nodes in Fig. 2 occupy 18 bytes, the new data structure in Fig. 4 requires 14 bytes for an internal node and 10 bytes for a terminal node.

There is the case with a very small probability that $(b - a)$ or $(a - foo)$ value is not less than 32767. In such a case, bar is set as 32767, and actual $(b - a)$ or $(a - foo)$ value is stored in an additional associative array. This array is composed of (key, data) pairs, and the pointer to the corresponding node serves as a key. Since a node can be either an internal or a terminal node, there is no duplication problem in the key values. There exist a lot of highly efficient implementations of the associative array such as hash_map in C++ Standard Template Library (STL). This array is attached to the end of the suffix subtree in the memory chunk when the subtree is stored on disk, and it is recreated by reading the attached list of pairs when the subtree is reloaded into main memory.

4. Evaluation

In this section, we evaluate our algorithm through a series of experiments. We compare (1) the size of the suffix tree for human genome sequences and (2) the time for suffix tree construction with the FAST algorithm. We used the same dataset as those in [1], [7]. We extracted the first 2, 5, 8, 11, 15, and 24 chromosomes and performed a separate experiment for each chromosome set. The size of the (concatenated) sequences were 441, 975, 1418, 1784, 2161, and 2701 MB, respectively. We set the prefix length as $p = 4$ for partitioning the suffixes.

The hardware platform was a PC equipped with Intel i7 2600 K 3.40 GHz CPU, 16 GB DDR3 main memory, and a 500 GB 7200 rpm hard disk. The software platform was Microsoft Windows 7 64 bit Edition, and Microsoft Visual C++ 2010 Express Edition was used as a C/C++ compiler. Although Intel i7 2600 K CPU has four cores, it can run up to eight parallel processes using Intel's Hyper-Threading Technology (HTT).

Figure 5 compares the size of the suffix trees constructed by FAST and our algorithm. The horizontal and vertical axes represent the size of genome sequences and the corresponding suffix trees, respectively. As a result of the

[†]The FAST algorithm [7] stored the chromosomes sequentially in a contiguous space. It was for improving disk access efficiency, and each chromosome was dealt with separately.

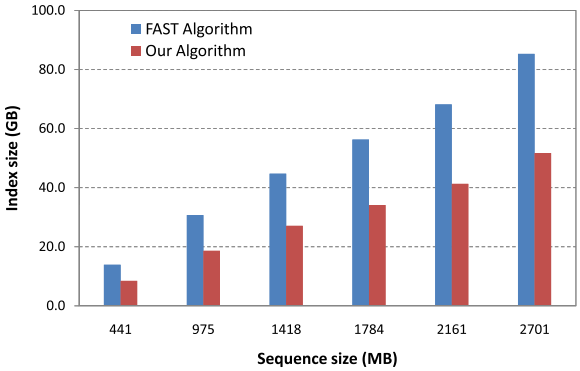


Fig. 5 Comparison of the suffix tree size: The suffix trees constructed by our algorithm occupy only 60% of the size compared to the FAST algorithm.

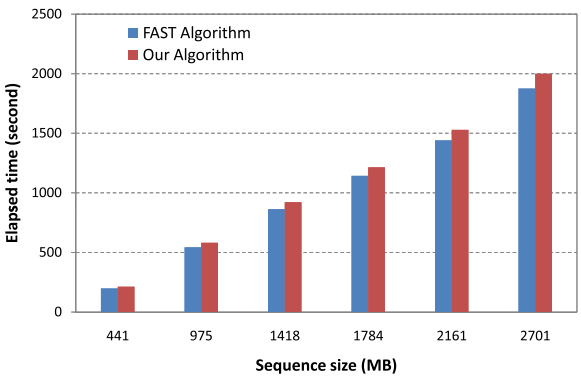


Fig. 6 Comparison of the construction time: Our algorithm needs only 6% of additional time compared to the FAST algorithm.

experiment, for every size of genome sequences, the suffix tree constructed by our algorithm occupied only about 60% of the size compared to FAST.

Figure 6 compares the time for constructing suffix trees by FAST and our algorithm. The vertical axis represents elapsed time in seconds. As a result of the experiment, our algorithm required only about 6% of additional time compared to FAST. The additional time was needed to compute the field values *b* or *foo* in Fig. 2 using the values of the field *bar* and the others in Fig. 4.

In conclusion, our algorithm constructed the suffix trees of a much smaller size for human genome sequences within almost the same time period as FAST. We believe that our algorithm should also construct small suffix trees even for the genome sequences of various organisms other than human beings.

References

[1] M. Barsky, U. Stege, A. Thomo, and C. Upton, "A new method for indexing genomes using on-disk suffix trees," Proc. ACM Conference on Information and knowledge Management (CIKM), pp.649–658, Napa Valley, California, Oct. 2008.

[2] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," Proc. Annual Symp. on Foundations of Computer Science (FOCS), Redondo Beach, California, pp.390–398, Nov. 2000.

[3] E. Hunt, M.P. Atkinson, and R.W. Irving, "Database indexing for large DNA and protein sequence collections," The VLDB Journal, vol.11, no.3, pp.256–271, 2002.

[4] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," Genome Biology, vol.10, no.3, pp.R25.1-R25.10, March 2009.

[5] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," Bioinformatics, vol.25, no.14, pp.1754–1760, July 2009.

[6] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: An improved ultrafast tool for short read alignment," Bioinformatics, vol.25, no.15, pp.1966–1967, Aug. 2009.

[7] W.-K. Loh, Y.-S. Moon, and W. Lee, "A fast divide-and-conquer algorithm for indexing human genome sequences," IEICE Trans. Inf. & Syst., vol.E94-D, no.7, pp.1369–1377, July 2011.

[8] B. Phoophakdee and M.J. Zaki, "Genome-scale disk-based suffic tree indexing," Proc. Int'l Conf. on Management of Data, ACM SIGMOD, pp.833–844, Beijing, China, June 2007.

[9] Y. Tian, S. Tata, R.A. Hankins, and J.M. Patel, "Practical methods for constructing suffix trees," VLDB Journal, vol.14, no.3, pp.281–299, 2005.