Efficient Implementations of Suffix Array Construction Algorithms

Sunglim Lee Kunsoo Park

School of Computer Science and Engineering,
Seoul National University,
Seoul, 151-744, Korea
{sllee,kpark}@theory.snu.ac.kr

Abstract

The suffix array is a general data structure that can be used to search DNA sequences in bioinformatics. In this paper we consider efficient implementations of three suffix array construction algorithms. They are the algorithms due to Manber-Myers [11], Kärkkäinen-Sanders [5] and Ko-Aluru [9]. The Kärkkäinen-Sanders and Ko-Aluru algorithms both run in linear time in the worst case. The Manber-Myers algorithm runs in linear time on average and $O(n \log n)$ time in the worst case. All three algorithms use linear space in the worst case. We developed several techniques to optimize implementations as much as possible. We conducted experiments to compare implementations of the three algorithms in terms of run-time and space. It is shown that the Kärkkäinen-Sanders and Ko-Aluru algorithms perform much better for DNA sequences than the Manber-Myers algorithm, which is competitive, however, for random inputs. A main interest in our experiments is to compare Ko-Aluru and Kärkkäinen-Sanders, and it is shown that Ko-Aluru performs better than Kärkkäinen-Sanders in our implementations.

Keywords. Suffix Array, DNA Sequence, Bioinformatics

1 Introduction

The suffix array is a data structure useful in searching a large text for matching substrings. In bioinformatics we often need to search DNA sequences, and suffix arrays are appropriate for searching DNA sequences for their efficiency and compactness. Suffix arrays are especially preferred to suffix trees [13, 15, 12, 10, 1, 2, 4, 14, 3] for their compactness.

In this paper we compare real-world performances of the following three suffix array construction algorithms.

- **Manber-Myers:** This suffix array construction algorithm uses the 'doubling technique' [11]. First, all the suffixes are grouped according to their first characters using bucket sort. Then, using the information of lexicographic order of suffixes up to the first characters, the suffixes are sorted up to the second characters. In the next step, the suffixes are sorted up to the fourth characters, the number of characters in the suffixes doubling in each step. The worst-case run-time of the algorithm is $O(n \log n)$, and the average-case run-time of the algorithm is suggested to be $O(n)$ [11]. In this paper, it

is shown that the input strings of DNA sequences act more as worst-case input for this algorithm. This algorithm uses space linear to the input size.

- **Kärkkäinen-Sanders:** This algorithm uses the approach of encoding, recursion and merging. Two-thirds of suffixes are encoded using their first three characters and sorted by recursion. The remaining $\frac{1}{3}$ suffixes are sorted and merged using the information of lexicographic order of the suffixes sorted by recursion. Thus the size of input reduces to $\frac{2}{3}$ on each recursion. Coupled with the fact that operations except recursion take linear time and space, the total run-time and space of this algorithm is $O(n)$.

- **Ko-Aluru:** Like the Kärkkäinen-Sanders algorithm, the Ko-Aluru algorithm uses the encoding-recursion-merging approach. The main difference between the two algorithms is the way the suffixes are encoded. In the Ko-Aluru algorithm, suffixes are classified as either type L or type S according to the lexicographic order between the first and the second character of the suffixes. Either type L or type S suffixes can be chosen to be encoded and sorted using recursion in each recursive call. By selecting the type with a smaller number of suffixes to be sorted recursively, the input size of each recursion is always less than $\frac{1}{2}$ of the previous input size. This is clearly a theoretical advantage over the Kärkkäinen-Sanders algorithm where the input size of each recursion is exactly $\frac{2}{3}$ of the previous input size. However, the merging of the Kärkkäinen-Sanders algorithm is simpler than that of the Ko-Aluru algorithm. Thus it is difficult to make a prediction on performances of these two algorithms and we need to conduct experiments to compare the two. The run-time and space of this algorithm is also $O(n)$.

Kim-Sim-Park-Park (KSPP). This algorithm [7] also uses the encoding-recursion-merging approach and runs in linear time and space. The KSPP algorithm divides the suffixes into the even suffixes and the odd suffixes and encodes one of them to be sorted recursively. Thus the input size for a recursion is exactly $\frac{1}{2}$ of the previous input size. However, because the merging step is too complex in the KSPP algorithm compared to those of Kärkkäinen-Sanders and Ko-Aluru, we chose not to include this algorithm in our comparison. In the KSPP algorithm, computing the longest common prefixes (LCP) between adjacent suffixes is an integral part of the algorithm.

We developed several techniques to optimize the implementations of these algorithms, and we conducted experiments to compare the three algorithms in terms of run-time and space. It is shown that the Kärkkäinen-Sanders and Ko-Aluru algorithms perform much better for DNA sequences than the Manber-Myers algorithm, but Manber-Myers is competitive for random inputs. A main interest in our experiments is to compare Ko-Aluru and Kärkkäinen-Sanders which are recently developed $O(n)$ algorithms, and it is shown that Ko-Aluru performs better than Kärkkäinen-Sanders with our implementations, contrary to the fact that Ko's own implementation of Ko-Aluru is worse than Kärkkäinen-Sanders [8]. The implementation issues we have considered will be explained in Section 2. The experimental results will be presented in Section 3 with some explanations on the results.

2 Implementation

In comparing the three algorithms, we tried to optimize the implementations as much as possible. We implemented the Ko-Aluru algorithm from scratch, and we used several techniques to get a high-performance implementation. For the Kärkkäinen-Sanders algorithm, we started with the source code implemented in C/C++ by Sanders, and we improved it by using techniques that we used for the Ko-Aluru algorithm. For the Manber-Myers algorithm, the source code implemented in C/C++ by Myers and Manber is used. All three implementations do not include codes for generating the LCP (longest common prefix) information. Once the sorted order of suffixes has been found, the LCP information between suffixes can be easily computed in linear time [6].

We focus on the cases where suffix array construction occurs entirely on-memory. The input string and the output suffix array are considered to reside on the main memory all the time as well as all intermediate working space used. The following are some techniques we used to build high-performance implementations.

2.1 Adapting to input alphabet size

The Manber-Myers implementation assumed the alphabet size of one byte, thus stored the input string as an array of bytes. This is well acceptable for DNA sequences because the alphabet size of a DNA sequence is less than a byte. The Ko-Aluru and Kärkkäinen-Sanders algorithms, on the other hand, are implemented using recursion. On both algorithms, the size of the alphabet increases exponentially as recursion goes. So even with an input string of a small alphabet size, the alphabet size will increase in each recursion, and the implementation should be able to handle a maximum alphabet size which is the same as the length of the input string, because the Ko-Aluru and Kärkkäinen-Sanders algorithms operate on integer alphabets. In this experiment, the maximum input string size is assumed to be 4 bytes, which is a reasonable value for both concurrent hardware environments and DNA sequences as input.

To make the Ko-Aluru implementation handle general-sized alphabets and at the same time prevent it from wasting memory by allocating memory for a 4-byte array for an input string where a 1-byte array will suffice, we implemented three versions of the recursion function where each takes 1byte, 2byte and 4byte input alphabets respectively. The recursion function is chosen from the three versions which best-fits the input alphabet size in run-time. Modifications are made to the Kärkkäinen-Sanders implementation in a similar way.

2.2 Reusing output memory on recursion

All three suffix array construction algorithm implementations take one input string and one output string to store the resulting suffix array on the memory space when the call to the implementations are made. Because the resulting suffix array is constructed at the very end of the algorithms, the space for the output suffix array can be used as working space during the algorithms. Moreover, in algorithms such as Ko-Aluru, this memory space can be used by the next recursive call by careful memory management. Modifications are made to the Kärkkäinen-Sanders implementation in a similar way.

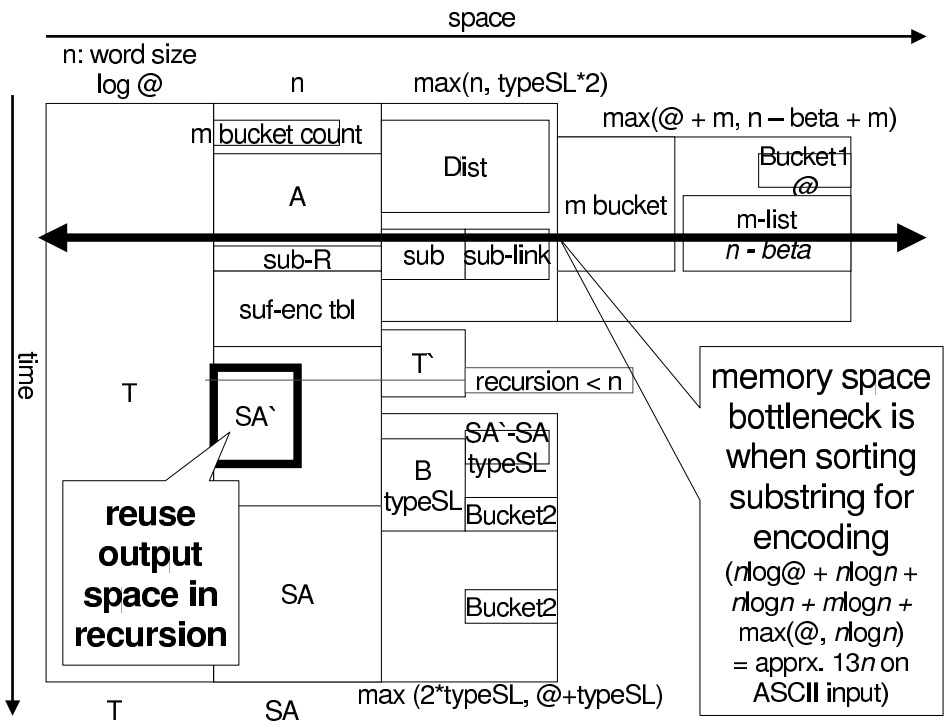

Figure 1: Memory map for implementation of Ko-Aluru

2.3 Sorting substrings in Ko-Aluru

As in the Kärkkäinen-Sanders algorithm, sorting substrings to be encoded for the next recursion is the most time consuming part in the Ko-Aluru algorithm. It is a more complicated problem in Ko-Aluru because in Ko-Aluru the substrings are of variable size whereas in Kärkkäinen-Sanders they are of fixed size. This part is the bottleneck of memory usage in the Ko-Aluru algorithm.

In our version of the Ko-Aluru algorithm, sorting the substrings is done using only one extra array other than the array of substrings and the corresponding reverse index array. We use repeated bucketing as stated in the Ko-Aluru algorithm, but we carefully store bucket indices, bucket counts and new bucket boundary information in a single array, still maintaining linear time operation.

2.4 Memory map for implementation of Ko-Aluru

We used a memory map shown in Figure 1 to schedule and optimize memory usage in implementing the Ko-Aluru algorithm. Ko-Aluru uses 18 different instances of data structures in a single recursion. Appropriate allocation and reusing of memory space for these data structures is essential to space-efficient implementation of Ko-Aluru.

The memory bottleneck in our implementation is shown in Figure 1 where 6 data structures are accessed simultaneously. Among the 6 data structures, T is the input string, *sub*, *sub-link* and *sub-r* store sorted substrings for further encoding, and *m-list* and *m-bucket* implement the *m-list* described in the Ko-Aluru algorithm [9]. This bottleneck memory space turns out to

be between $13n$ and $15n$ for an n-byte input when implemented using a 32 bit machine where one word is 4 bytes. Our measured memory usage shown in the following section confirms this statement.

3 Results

3.1 Runtime environment

The experiments are carried out using LINUX machine with dual intel 2.4MHz CPU and 2GB of memory. The run-time is measured using the system clock. Each measurement was taken at least 5 times to reduce inaccuracy due to LINUX being a multiprocessing OS. The memory space used is measured in terms of maximum process memory allocated during execution.

3.2 Data

We used three kinds of data. They are 5 DNA sequences, 1 text data that contains the Bible and 10 randomly generated strings of various lengths and alphabet sizes. The DNA sequences have an alphabet of 4 characters, namely A, G, T, C. A randomly generated string has an alphabet size of either 6 or 86.

3.3 Run-time

Table 1 shows measured run-times of the implementations on various inputs. Figure 2 compares the run-times with all values normalized to the Ko-Aluru implementation, so that we can easily see the relative performance of each algorithm on different types of input. Notice that the performances of the Manber-Myers algorithm on DNA sequences and random strings show a notable difference. The Manber-Myers algorithm is clearly slow with DNA sequences as input.

Figure 3 picks some cases from Table 1 with similar input lengths of about 5,000,000. More clearly it can be seen that the Manber-Myers algorithm runs slower with DNA sequence as input in absolute sense when compared to random strings of the same size. The Ko-Aluru algorithm is shown to be quite efficient with DNA sequences.

3.4 Memory

Figure 4 shows the memory usage of each algorithm. The memory usage of various types of inputs is shown. It can be observed that the amount of memory used by each algorithm is only dependent on the input length and not dependent on the type of data. Recall that in the Ko-Aluru algorithm, the input sizes of successive recursive calls are not fixed and are sensitive to the type of input data. However, such behavior in memory usage is not observed in our results because the temporary working-space memory required by the first recursion always exceeds the sum of memory used by all the subsequent recursions.

The Manber-Myers algorithm uses less than $10n$ bytes when the input string size is n bytes. The Ko-Aluru and Kärkkäinen-Sanders algorithms both use less than $14n$ bytes for n size input, which is a little more than Manber-Myers, but it is still far less than suffix trees to justify using suffix arrays over suffix trees.

Figure 5 shows memory usage improvements made by our modifications on the original implementation by Sanders.

Input String				Run-time (sec)		
Name	Type	Length	Alpha-bet	Ko-Aluru	Kärkkäinen-Sanders	Manber-Myers
NT032977.5	DNA sequence	16,508,532	4	32.66	64.75	153.54
NC002927	DNA sequence	5,339,179	4	9.94	21.45	64.92
NC002928	DNA sequence	4,773,551	4	8.52	18.89	64.37
NC002929	DNA sequence	4,086,189	4	7.12	15.90	54.03
NC005061	DNA sequence	705,557	4	0.76	2.02	4.37
BibleKJV.txt	English text	4,347,404	70	8.71	17.50	35.15
10M-6	Random string	10,000,000	6	21.77	27.67	33.19
5M-6	Random string	5,000,000	6	10.38	13.08	16.05
1M-6	Random string	1,000,000	6	1.52	2.39	2.97
500K-6	Random string	500,000	6	0.55	1.05	1.36
100K-6	Random string	100,000	6	0.05	0.06	0.11
100M-6	Random string	100,000,000	6	253.62	383.44	500.25
10M-86	Random string	10,000,000	86	30.53	27.07	33.26
1M-86	Random string	1,000,000	86	2.38	2.45	1.74
500K-86	Random string	500,000	86	0.84	1.11	1.30
100K-86	Random string	100,000	86	0.06	0.08	0.11

Table 1: Run-time comparison of suffix array construction

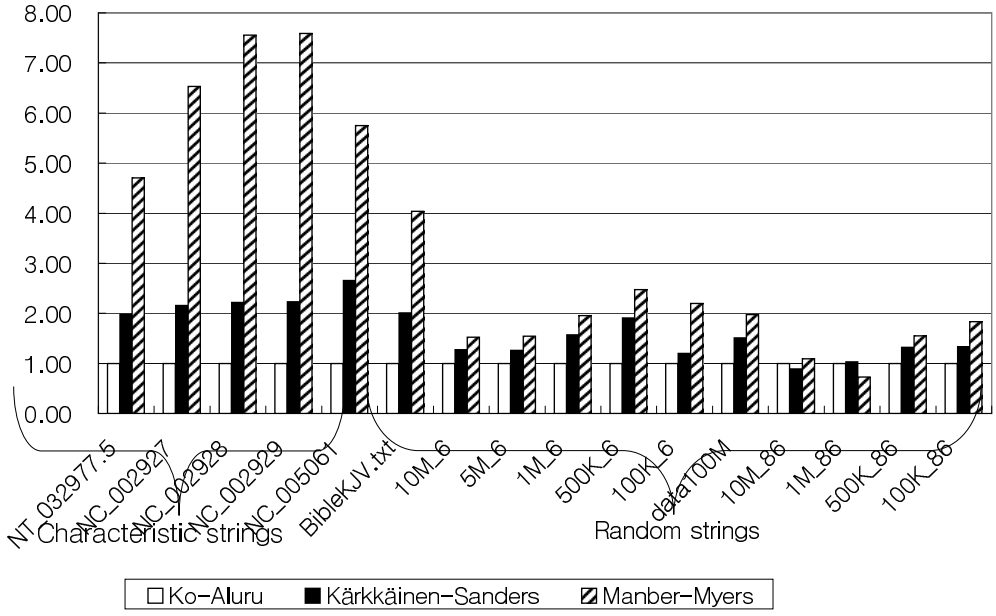

Figure 2: Comparison of run-time, normalized on Ko-Aluru

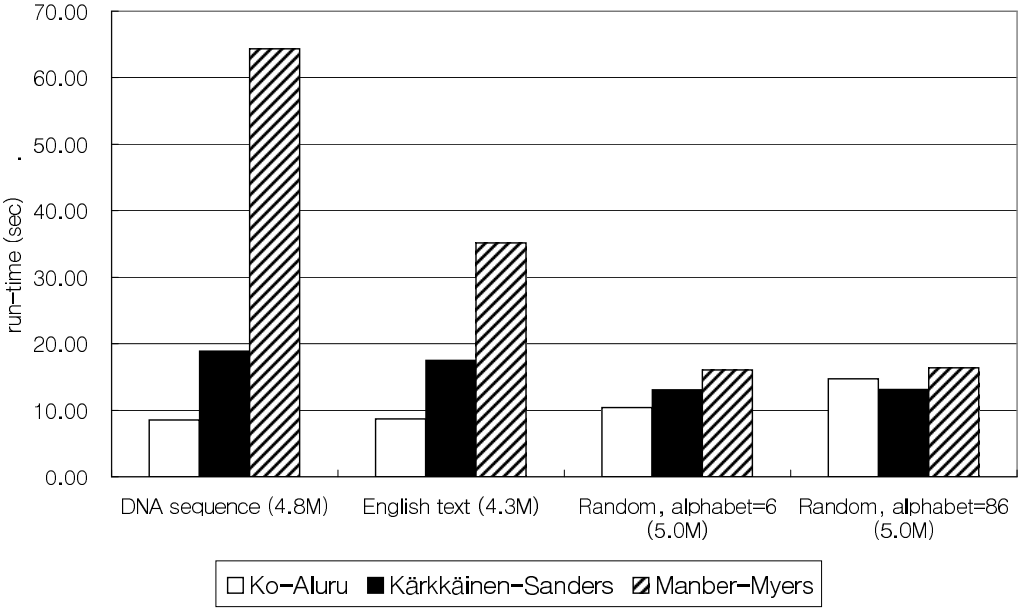

Figure 3: Comparison of run-time on inputs with similar length

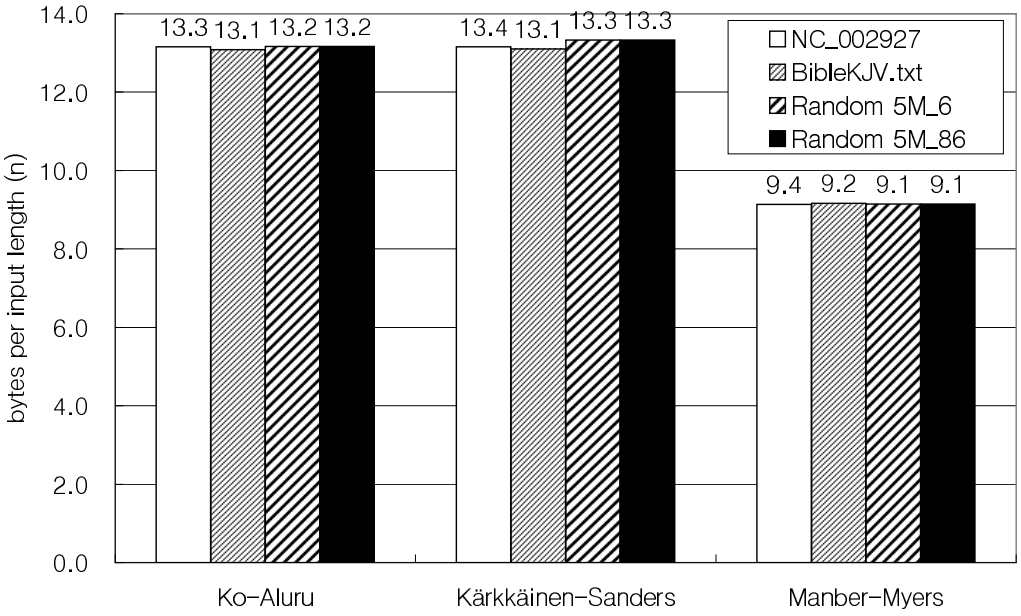

Figure 4: Comparison of memory usage

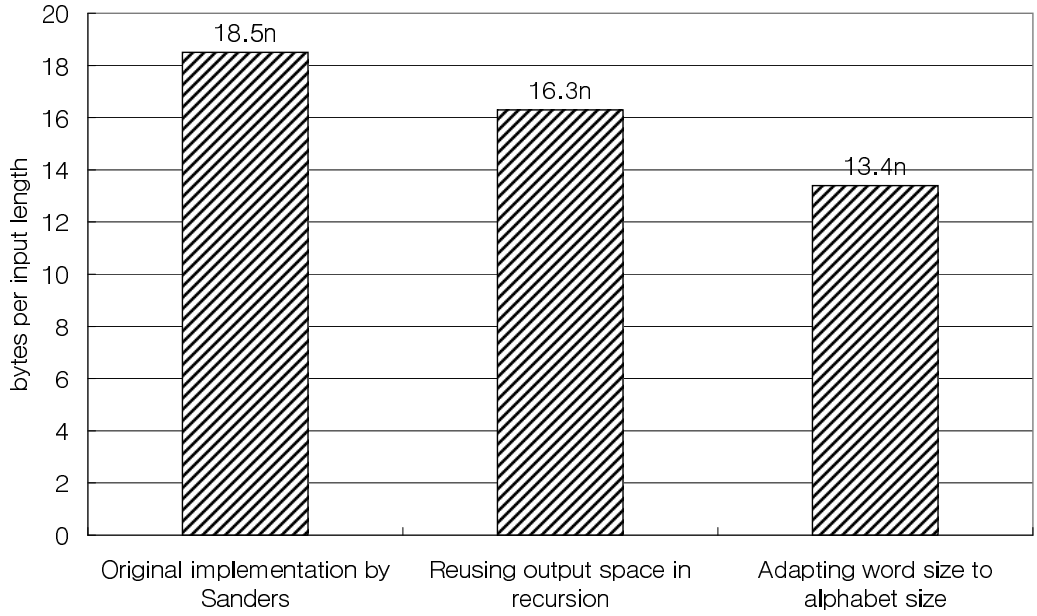

Figure 5: Memory usage reduced by modifying Kärkkäinen-Sanders implementation

4 Conclusions

In this paper we showed that the Ko-Aluru algorithm performed best for creating suffix arrays for DNA sequences. Kärkkäinen-Sanders took about twice the time than Ko-Aluru for DNA sequences. The performance of Manber-Myers was far behind, which took 4 to 8 times longer than Ko-Aluru. The working space required was almost the same for Ko-Aluru and Kärkkäinen-Sanders. Manber-Myers required about 70% of the space used by Ko-Aluru or Kärkkäinen-Sanders, so Manber-Myers can be a viable option in situations where reducing memory usage is most important. However, the performance of the three algorithms differed much less for random inputs. Though Ko-Aluru still performed best for random inputs, the differences were minor.

The Ko-Aluru algorithm is more suitable when the input string lacks randomness. Ko-Aluru encodes a problem into a subproblem of variable size and the input size on recursion is likely to be far less than $\frac{1}{2}$ of the previous input length when the input is not random. The Kärkkäinen-Sanders algorithm, however, cannot enjoy this advantage because the input size on recursion is always fixed to be $\frac{2}{3}$ of the previous input length.

We conclude that the Ko-Aluru algorithm is preferable to the Kärkkäinen-Sanders algorithm or the Manber-Myers algorithm when the input string is a DNA sequence.

References

[1] S. Burkhardt and J. Karkkainen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symposium on Combinatorial Pattern Matching. LNCS 2676, Springer, 2003*, pages 55-69.

[2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science*, 390-398, 2000.

[3] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. Proc. of ACM Symposium on Theory of Computing, pp. 397-406, 2000.

[4] W. Hon, K. Sadakane and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. of the 44st IEEE Symposium on Foundations of Computer Science*, 251-260, 2003.

[5] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*, Springer, 2003.

[6] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *12th Annual Symposium on Combinatorial Pattern Matching*, July 2001, Lecture Notes in Computer Science, vol.2089, Springer, 2001, pp.181-192.

[7] D. K. Kim, J. S. Sim, H. Park and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, Springer, June 2003.

[8] P. Ko. Private communication.

[9] P. Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, Springer, June 2003.

[10] N.J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

[11] U. Manber and G. Myers. Suffix array: A new method for on-line string searches. In *Proc. of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319-327, 1990.

[12] G. Manzini and P. Ferragina. Engineering a Lightweight Suffix Array Construction Algorithm. In *Proc. of European Symposium on Algorithms 2002.*

[13] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262-272, 1976.

[14] K. Sadakane. Succint representation of lcp information and improvement in the compressed suffix arrays. ACM-SIAM Symp. on Discrete Algorithms, 225-232, 2002.

[15] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249-260, 1995.