Checking Big Suffix and LCP Arrays by Probabilistic Methods

Yi Wu, Ge Nong, *Senior Member, IEEE*, Wai Hong Chan, and Ling Bo Han

Abstract—For full-text indexing of massive data, the suffix and LCP (longest common prefix) arrays have been recognized as fundamental data structures, and there are at least two needs in practice for checking their correctness, i.e., program debugging and verifying the arrays constructed by probabilistic algorithms. Two probabilistic methods are proposed to check the suffix and LCP arrays of constant or integer alphabets in external memory using a Karp-Rabin fingerprinting technique, where the checking is wrong only with a negligible error probability. The first method checks the lexicographical order and the LCP-value of two suffixes by computing and comparing the fingerprints of their LCPs. This method is general in terms of that it can verify any full or sparse suffix/LCP array of any order. The second method uses less space, it first employs the fingerprinting technique to verify a subset of the given suffix and LCP arrays, from which two new suffix and LCP arrays are induced and compared with the given arrays for verification, where the induced suffix and LCP arrays can be removed for constant alphabets to save space.

Index Terms—Suffix and LCP arrays, verification, karp-rabin fingerprinting, external memory

1 Introduction

1.1 Background

Suffix and longest common prefix (LCP) arrays play an important role in various string processing tasks, such as data compression, pattern matching and genome assembly. In many applications, these two data structures make up the core part of a powerful full-text index, called enhanced suffix array [1], which is more space efficient than a suffix tree and applicable to emulating most searching functionalities provided by the latter in the same time complexity. The first algorithm for building suffix array (SA) in internal memory was presented in [2]. From then on, much more effort has been put on designing efficient constructors for suffix array on different computation models [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. In respect of the research on LCP array construction algorithms, the existing works can be classified into two categories with regard to their input requirements, where the algorithms from the first category compute both suffix and LCP arrays at the same time with the original text only [10], [14], [15], and those from the second category carry out the computation by taking SA and/or Burrows-Wheeler transform (BWT) as additional inputs [14], [16], [17], [18], [19]. So far, the algorithms designed by the induced sorting (IS) principle take linear time and space to run and get the best results on both internal and external memory [6], [10]. In addition to the sequential algorithms, there are also parallel algorithms proposed to achieve high performance by fully using the available multi-core CPUs and/or GPUs [19], [20], [21], [22], [23].

While the research on efficient construction of suffix and LCP arrays keeps evolving, the algorithms proposed recently are becoming more complicated than before. Currently, the open source programs for the state-of-the-art algorithms are provided "as-is" for demonstration and experiment purpose only, giving no guarantee that they have correctly implemented the algorithms. As a common practice, a suffix or LCP checker is provided to check the correctness of a constructed array. For example, such a checker can be found in some software packages for DC3 [24], SA-IS [6], eSAIS [10] and so forth. In addition to help avoid implementation bugs, a checker is also demanded for an array constructed by a probabilistic algorithm (e.g., [25]). In this case, the array is correctly constructed with a probability and hence must be verified by a checker to ensure its correctness. As far as we know, the work in [26] describes the only SA checking method that can be found in the existing literature, and no efficient checking method for LCP array has been reported yet. Particularly, there is currently no reported solution that can check both the suffix and the LCP arrays in external memory. This motivates our work here to design efficient checkers for big suffix and LCP arrays in external memory.

1.2 Contribution

Our contribution comprises two methods to probabilistically verify any given suffix and LCP arrays. In principle, Method

• *Y. Wu and G. Nong are with the Department of Computer Science, Sun Yat-sen University, Guangzhou, and the SYSU-CMU Shunde International Joint Research Institute, Shunde 528000, China.*
E-mail: wu.yi.christian@gmail.com, issng@mail.sysu.edu.cn.
• *W. H. Chan is with the Department of Mathematics and Information Technology, The Education University of Hong Kong, Tai Po 4600-652, Hong Kong. E-mail: waihchan@ied.edu.hk.*
• *L. B. Han is with the Key Laboratory of Machine Intelligence and Advanced Computing (Sun Yat-sen University), Ministry of Education, Guangzhou, China. E-mail: hanlb@mail2.sysu.edu.cn.*

Manuscript received 26 Oct. 2016; revised 24 Apr. 2017; accepted 28 Apr. 2017. Date of publication 8 May 2017; date of current version 14 Sept. 2017.
(Corresponding author: G. Nong and W. H. Chan.)
Recommended for acceptance by A. Guzman-Arenas
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2702642

A checks the lexical order and the LCP-value of two neighboring suffixes in the suffix array by literally comparing the characters of their LCPs. For reducing the time complexity of a comparison between two sequences of characters, we use a Karp-Rabin fingerprinting technique to convert each sequence into a single integer, called fingerprint, and compare the fingerprints instead to check the equality of two sequences. The algorithm for Method A involves multiple scans and sorts on sets of $\mathcal{O}(n)$ fixed-size items. Its implementation in external memory suffers from a space bottleneck due to the large disk volume taken by each sort. To overcome this drawback, Method B first employs the fingerprinting technique to check a subset selected from the given suffix and LCP arrays, then it utilizes the IS method to produce the final suffix and LCP arrays from the verified subset and literally compares them with the input arrays to ensure the correctness of the latter. Our experiments indicate that the program for Algorithm 2 designed by Method B only takes around half as much disk space as the program for Algorithm 1 designed by Method A.

The remainder of this paper is organized as follows. Sections 2 and 3 describe the two methods and their algorithmic designs. Section 4 conducts an experimental study for performance evaluation of our programs for these algorithms. Finally, Section 5 gives some concluding remarks.

2 Method A

2.1 Preliminaries

Given a string $x[0, n-1]$ drawn from a constant or integer alphabet Σ of size $\mathcal{O}(1)$ or $\mathcal{O}(n)$, respectively, the suffix array of x, denoted by sa, is a permutation of $\{0, 1, \dots, n-1\}$ such that $\mathsf{suf}(sa[i]) < \mathsf{suf}(sa[j])$ for $i, j \in [0, n)$ and $i < j$, where $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ are two suffixes starting with $x[sa[i]]$ and $x[sa[j]]$, respectively. Particularly, we say $\mathsf{suf}(sa[j])$ is a lexical neighbor of $\mathsf{suf}(sa[i])$ if $|i-j| = 1$. The LCP array of x, denoted by lcp, consists of n integers, where $lcp[0] = 0$ and $lcp[i]$ records the LCP-value of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[i-1])$ for $i \in [1, n)$. A suffix/LCP array is infinite-order if the suffixes/LCPs are sorted/counted up to their ends, respectively, or else finite-order.

2.2 Idea

According to the above definitions, we give in Lemma 1 the sufficient and necessary conditions for checking both suffix and LCP arrays. Notice that the lexical order and the LCP-value of any two suffixes in x can be computed by literally comparing their characters. For convenience, we append a virtual character to x and assume it to be lexicographically smaller than any characters in Σ, hence any two suffixes are different.

Lemma 1. *Both $sa[0, n)$ and $lcp[0, n)$ are correct if and only if the following conditions are satisfied, for all $i \in [1, n)$:*

(1) *sa is a permutation of* $\{0, 1, \dots, n-1\}$.
(2) $x[sa[i], sa[i] + lcp[i] - 1] = x[sa[i-1], sa[i-1] + lcp[i] - 1]$.
(3) $x[sa[i] + lcp[i]] > x[sa[i-1] + lcp[i]]$.

Proof. Both the sufficiency and necessity are immediately from the definitions of suffix and LCP arrays. Condition 1 guarantees that each suffix is in sa, conditions 2 and 3 guarantees that the lexical order and the LCP-value of any two neighboring suffixes in sa are both correct. □

Directly comparing the characters of two suffixes to determine their LCP has the worst case time of $O(n)$. An alternative is to exploit a perfect hash function to convert each substring into a single integer such that any two substrings have a common hash value if and only if they are literally equal to each other, hence the hash values of two substrings can be compared instead to check the equality of two substrings. Taking into account the high difficulty of finding a perfect hash function to meet this requirement, we prefer using a Karp-Rabin fingerprinting function [27] to transform a substring into an integer called fingerprint. To be specific, suppose L is a prime and δ is a number randomly chosen from $[1, L)$, the fingerprint $\mathsf{fp}(i, j)$ for a substring $x[i, j]$ can be iteratively calculated according to the formulas below: Scan x rightward to iteratively compute $\mathsf{fp}(0, k)$ for all $k \in [0, n)$ using Formulas 1-2, record $\mathsf{fp}(0, i-1)$ and $\mathsf{fp}(0, j)$ during the calculation and subtract the former from the latter to obtain $\mathsf{fp}(i, j)$ using Formula 3.

Formula 1. $\mathsf{fp}(0, -1) = 0$.

Formula 2. $\mathsf{fp}(0, i) = \mathsf{fp}(0, i-1) \cdot \delta + x[i] \bmod L$ for $i \geq 0$.

Formula 3. $\mathsf{fp}(i, j) = \mathsf{fp}(0, j) - \mathsf{fp}(0, i-1) \cdot \delta^{j-i+1} \bmod L$.

Notice that two equal substrings always share a common fingerprint, but the inverse is not true. The probability of a false match can be reduced to a negligible level by setting L to a large value [27], this property is utilized in [25] to design a probabilistic algorithm for computing a sparse suffix array. Hence we have:

Corollary 1. *Both $sa[0, n)$ and $lcp[0, n)$ are correct with a high probability given these conditions, for all $i \in [1, n)$:*

(1) *sa is a permutation of* $\{0, 1, \dots, n-1\}$.
(2) $\mathsf{fp}(sa[i], sa[i] + lcp[i] - 1) = \mathsf{fp}(sa[i-1], sa[i-1] + lcp[i] - 1)$.
(3) $x[sa[i] + lcp[i]] > x[sa[i-1] + lcp[i]]$.

Fig. 1 gives an illustrating example for utilizing Corollary 1 to check the input suffix and LCP arrays. Given that $L = 197$ and $\delta = 101$, lines 4-8 compute $\mathsf{fp}(0, p)$ iteratively according to Formulas 1-2. Lines 10-20 use these values to compute the fingerprints for all the target substrings. In more detail, consider the leftmost pair of neighboring suffixes in sa, i.e., $\mathsf{suf}(sa[0])$ and $\mathsf{suf}(sa[1])$, the substrings given by their LCP-value are $x[sa[0], sa[0] + lcp[1] - 1]$ and $x[sa[1], sa[1] + lcp[1] - 1]$, respectively. According to Formula 3, $\mathsf{fp}(sa[0], sa[0] + lcp[1] - 1)$ is equal to the difference between $\mathsf{fp}(0, sa[0] - 1)$ and $\mathsf{fp}(0, sa[0] + lcp[1] - 1)$, both have been calculated. Following the same way, $\mathsf{fp}(sa[1], sa[1] + lcp[1] - 1)$ is computed by reducing $\mathsf{fp}(0, sa[1] - 1)$ from $\mathsf{fp}(0, sa[1] + lcp[1] - 1)$. Hence, we obtain the fingerprints for these two substrings in lines 10-14 and see that they are equal to each other.

2.3 Algorithm

We describe an approach for checking the conditions in Corollary 1 on random access models, of which the core

```
       p:   0   1   2   3   4   5   6   7   8   9  10  11  12  13
    x[p]:   2   1   3   1   3   1   2   1   3   1   3   1   2   1
   sa[p]:  13  11   5   9   3   7   1  12   6   0  10   4   8   2
  lcp[p]:   0   1   3   1   5   3   7   0   2   8   0   4   2   6
Compute fp(0, p) for p ∈ [0, n):
            fp(0,0) = fp(0,−1) · 101 + x[0] mod 197 = 2,
            fp(0,1) = fp(0,0) · 101 + x[1] mod 197 = 6,
            fp(0,2) = fp(0,1) · 101 + x[2] mod 197 = 18,
            ......
fp(0, p):   2   6  18  46 118  99 151  83 112  84  16  41   6  16
For suf(sa[0]) and suf(sa[1]):
            fp(sa[1], sa[1] + lcp[1] − 1) = fp(11) − fp(10) · 101^1 mod 197
            = 1
            fp(sa[0], sa[0] + lcp[1] − 1) = fp(13) − fp(12) · 101^1 mod 197
            = 1
For suf(sa[1]) and suf(sa[2]):
            fp(sa[2], sa[2] + lcp[2] − 1) = fp(7) − fp(4) · 101^3 mod 197
            = 160
            fp(sa[1], sa[1] + lcp[2] − 1) = fp(13) − fp(10) · 101^3 mod 197
            = 160
            ......
```

Fig. 1. An example for computing and comparing fingerprints for substrings specified by the LCP-values of neighboring suffixes in sa.

part is to check the lexical order and the LCP-value for each pair of neighboring suffixes in sa during the scan of sa and lcp. This is done by using Formulas 1-3. Two zero-initialized arrays fp and mk are introduced to facilitate the checking process, where fp is for storing the fingerprints of all the prefixes in x and mk is for checking whether sa stores a permutation of $\{0, 1, \ldots, n-1\}$.

S1 Scan x with i increasing from 0 to $n-1$, for each scanned $x[i]$, compute $\mathsf{fp}(0, i)$ and assign to $fp[i]$.

S2 Scan sa and lcp with i increasing from 1 to $n-1$, for each scanned $sa[i]$ and $lcp[i]$, let $u = sa[i], v = lcp[i]$ and $w = sa[i-1]$, perform:
  (a) Retrieve $fp[u-1]$ and $fp[u+v-1]$ from fp to compute $\mathsf{fp}(u, u+v-1)$, set $mk[u] = 1$;
  (b) Retrieve $fp[w-1]$ and $fp[w+v-1]$ from fp to compute $\mathsf{fp}(w, w+v-1)$;
  (c) Check if $\mathsf{fp}(u, u+v-1) = \mathsf{fp}(w, w+v-1)$ and $x[u+v] > x[w+v]$;
  (d) Set $mk[sa[0]] = 1$.

S3 Check if $mk[i] = 1$ for all $i \in [0, n)$.

The above approach takes $\mathcal{O}(n)$ time and space in internal memory. The difficulty for applying it in external memory is step S2, it suffers from a performance degradation caused by frequent random accesses to disks. Assume that x, sa and lcp are stored in external memory, we design Algorithm 1 for conducting these I/O operations in a disk-friendly way. The idea is to first sort data in the order that they are to be visited and then access them sequentially. For this purpose, Algorithm 1 first scans sa and lcp to produce ST_1, ST_2, ST_3, and sorts their tuples by the first components in ascending order at the beginning (lines 2-5). Afterward, it iteratively computes the fingerprints of all the prefixes of x according to Formulas 1-2 and assigns them to the sorted tuples as follows (lines 6-21): When figuring out $\mathsf{fp}(0, i-1)$, extract each tuple e with $e.1\text{st} = i$ from $ST_1/ST_2/ST_3$, update e with $\mathsf{fp}(0, i-1)$, and then forward e to $ST'_1/ST'_2/ST'_3$. Because the first components of the tuples in ST_1 constitute a copy of sa, the algorithm checks condition 1 when scanning these tuples in their sorted order. Finally, it sorts the updated tuples back to their original order (line 22) and visits them sequentially to check conditions 2 and 3 following the same way of step S2 (lines 23-31).

Algorithm 1. The Algorithm Based on Corollary 1

```
1:  Function CheckByFP($x$, $sa$, $lcp$, $n$)
2:      $ST_1 := [(sa[i], i, null) | i \in [0, n)]$
3:      $ST_2 := [(sa[i] + lcp[i+1], i, null, null) | i \in [0, n-1)]$
4:      $ST_3 := [(sa[i] + lcp[i], i, null, null) | i \in [1, n)]$
5:      sort the tuples in $ST_1$, $ST_2$ and $ST_3$ by the 1st
        components, respectively;
6:      $fp := 0$
7:      for $i \in [0, n]$ do
8:          if $ST_1$.notEmpty() and $ST_1$.top().$1st = i$ then
9:              $e := ST_1$.top(), $ST_1$.pop(), $e.3rd := fp$,
                $ST'_1$.push($e$)
10:         end
11:         else
12:             return false        // condition 1 is violated
13:         end
14:         while $ST_2$.notEmpty() and $ST_2$.top().$1st = i$ do
15:             $e := ST_2$.top(), $ST_2$.pop(), $e.3rd := fp$,
                $e.4th := x[i]$, $ST'_2$.push($e$)
16:         end
17:         while $ST_3$.notEmpty() and $ST_3$.top().$1st = i$ do
18:             $e := ST_3$.top(), $ST_3$.pop(), $e.3rd := fp$,
                $e.4th := x[i]$, $ST'_3$.push($e$)
19:         end
20:         $fp := fp \cdot \delta + x[i] \bmod P$     // $x[n]$ is the virtual
                                                      character
21:     end
22:     sort the tuples in $ST'_1$, $ST'_2$ and $ST'_3$ by the 2nd
        component, respectively;
23:     for $i \in [1, n)$ do
24:         $fp_1 := ST'_1$.top().$3rd$, $ST'_1$.pop(), $fp_2 := ST'_2$.top().$3rd$,
            $ch_1 := ST'_2$.top().$4th$, $ST'_2$.pop()
25:         $\hat{fp_1} = fp_2 - fp_1 \cdot \delta^{lcp[i]} \bmod P$
26:         $fp_1 := ST'_1$.top().$3rd$, $fp_3 := ST'_3$.top().$3rd$,
            $ch_2 := ST'_3$.top().$4th$, $ST'_3$.pop()
27:         $\hat{fp_2} = fp_3 - fp_1 \cdot \delta^{lcp[i]} \bmod P$
28:         if $\hat{fp_1} \neq \hat{fp_2}$ or $ch_1 \geq ch_2$ then
29:             return false        // condition 2 or 3 is violated
30:         end
31:     end
32:     return true
```

The last point is how to obtain $\delta^{lcp[i]}$ quickly when computing $\hat{fp_1}$ and $\hat{fp_2}$ in lines 25 and 27. One way is to keep a lookup table in internal memory to store all $\delta^{lcp[i]}$. This can answer the question in constant time, but it is space-consuming and impractical to be used in external memory. Notice that the LCP of any two suffixes is shorter than n, we can return the answer in $\mathcal{O}(\lceil \log_2 n \rceil)$ time using $\mathcal{O}(\lceil \log_2 n \rceil)$ internal memory. Let e be an integer from $[0, n)$, its binary form is $k_{\lceil \log_2 n \rceil} \ldots k_1 k_0$. We have $\delta^e = \Pi_{i=0}^{\lceil \log_2 n \rceil} \delta^{k_i \cdot 2^i}$, which can be computed with $\{\delta^1, \delta^2, \ldots, \delta^{2^{\lceil \log_2 n \rceil}}\}$ already known.

2.4 Analysis

Algorithm 1 performs multiple scans and sorts on the arrays of $\mathcal{O}(n)$ fixed-size tuples in disks. Given RAM size M, disk size D and block size B, all are in words, the time and I/O complexities for each scan are $\mathcal{O}(n)$ and $\mathcal{O}(n/B)$, respectively, while those for each sort are $\mathcal{O}(n\log_{M/B}(n/B))$ and $\mathcal{O}((n/B)\log_{M/B}(n/B))$, respectively [28]. Algorithm 1 reaches its peak disk use when sorting the tuples in lines 5 and 22. Suppose the input string and the suffix/LCP array are encoded as α- and β-byte integers, respectively, and each fingerprint is a γ-byte integer, it takes $(2 \cdot \beta + \gamma)$ space for sorting the tuples of ST_1/ST_1' and $(\alpha + 2 \cdot \beta + \gamma)$ for sorting the tuples of ST_2/ST_2' and ST_3/ST_3'. For saving space, our program implementing the algorithm tackles ST_1/ST_1', ST_2/ST_2' and ST_3/ST_3' separately and performs a single scan over x for each of them to obtain the fingerprints, using less space but more time. The experiment in Section 4 indicates that the disk use is 40 times of x.

3 METHOD B

3.1 Preliminaries

We further to give another checking method using the induced sorting principle [6], [24], which requires much less space than Method A. For presentation convenience, we introduce some symbols and notations as below.

Character and suffix classification. All the characters in x are classified into three types, namely L-, S- and S*-type. In detail, $x[i]$ is L-type if (1) $i = n-1$ or (2) $x[i] > x[i+1]$ or (3) $x[i] = x[i+1]$ and $x[i+1]$ is L-type; otherwise, $x[i]$ is S-type. Further, if $x[i]$ and $x[i+1]$ are separately L-type and S-type, then $x[i+1]$ is also S*-type. Moreover, a suffix is L-, S- or S*-type if its heading character is L-, S-, or S*-type, respectively.

Suffix and LCP buckets. All the suffixes in sa are partitioned into multiple buckets and those of a common heading character are grouped into a single bucket that occupies a contiguous interval in sa. Each bucket can be further divided into two sub-buckets, where the left and the right parts contain L- and S-type suffixes only, respectively. For short, we use $\mathsf{sa\_bkt}(c)$ to denote the bucket storing the suffixes starting with character c and $\mathsf{sa\_bkt_L}(c)/\mathsf{sa\_bkt_S}(c)$ to denote its left/right sub-bucket. Accordingly, lcp can be also split into multiple buckets, where $\mathsf{lcp\_bkt}(c)/\mathsf{lcp\_bkt_L}(c)/\mathsf{lcp\_bkt_S}(c)$ stores the LCP-values of suffixes in $\mathsf{sa\_bkt}(c)/\mathsf{sa\_bkt_L}(c)/\mathsf{sa\_bkt_S}(c)$, respectively.

Suffix and LCP arrays for S*-type suffixes. Given that the number of S*-type suffixes is n_1, $sa^*[0, n_1)$ stores all the S*-type suffixes arranged in lexical order, while $lcp^*[0] = 0$ and $lcp^*[i]$ records the LCP-value of $\mathsf{suf}(sa^*[i])$ and $\mathsf{suf}(sa^*[i-1])$ for $i \in [1, n_1)$.

Type Array. The array t records in $t[i]$ the type information of $x[i]$, where $t[i] = 1$ or 0 if $x[i]$ is S- or L-type, respectively.

3.2 Idea

The induced sorting principle has been extensively used to design efficient algorithms for constructing the suffix and LCP arrays in internal or external memory [6], [10], [12], [13], [14]. Such a construction algorithm mainly consists of a reduction phase for computing sa^* and lcp^*, followed by an induction phase for inducing sa and lcp from sa^* and lcp^*[1], which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2017.2702642. Given that sa^* and lcp^* are already known, we can induce the final suffix and LCP arrays from them. This suggests a checking method based on Lemma 2.

Lemma 2. *Both $sa[0, n)$ and $lcp[0, n)$ are correct if and only if the conditions below are satisfied:*

(1) Both sa^* and lcp^* are correct.

(2) $sa = sa'$ and $lcp = lcp'$, where sa' and lcp' are induced from sa^* and lcp^* by the IS method.

We have Corollary 2 for probabilistically checking the conditions of Lemma 2.

Corollary 2. *Both $sa[0, n)$ and $lcp[0, n)$ are correct with a high probability given the following conditions, for all $i \in [1, n_1)$ and $j \in [0, n)$:*

(1) $x[sa^*[i]]$ is S-type, and $sa^*[i] \neq sa^*[k]$ for all $k \in [0, n_1)$ and $k \neq i$.*

(2) $\mathsf{fp}(sa^*[i], sa^*[i] + lcp^*[i] - 1) = \mathsf{fp}(sa^*[i-1], sa^*[i-1] + lcp^*[i] - 1)$.

(3) $x[sa^*[i] + lcp^*[i]] > x[sa^*[i-1] + lcp^*[i]]$.

(4) $sa[j] = sa'[j]$ and $lcp[j] = lcp'[j]$ for $j \in [0, n)$, where sa' and lcp' are induced from sa^* and lcp^* by the IS method.

3.3 Algorithm

We further to design Algorithm 2 for checking the conditions of Corollary 2. The first step is to compute and verify sa^* and lcp^*. Similar to Method A, the fingerprinting technique is employed to probabilistically check the correctness of sa^* and lcp^*. The array sa^* can be produced by sequentially retrieving the S*-type suffixes from sa and the LCP-value of two successive S*-type suffixes in sa, say $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$, is equal to the minimal of $\{lcp[i+1], \ldots, lcp[j-1], lcp[j]\}$. The algorithm first sorts all the suffixes in sa by their starting positions (lines 2-3) and then scans x once to get the S*-type suffixes (lines 4-13). After that, it puts these S*-type suffixes back in their lexical order and outputs them one by one to generate sa^* (lines 14-27). Meanwhile, it calculates the LCP-value for each pair of the neighboring suffixes in sa^* by tracing the minimum in the lcp interval between these two suffixes. Notice that, we check condition 1 when visiting the suffixes in their position order (lines 7-12), and check conditions 2 and 3 by Algorithm 1 with sa^* and lcp^* as input. Given sa^* and lcp^* are correct, Algorithm 2 invokes an inducing process by employing the IS method to induce sa' and lcp' from sa^* and lcp^* (line 31) and compares them with sa and lcp to complete the whole checking process (lines 32-36).

Assume that the alphabet Σ is of size $\mathcal{O}(1)$, we can check sa and lcp without storing the induced suffix and LCP arrays in Algorithm 2. The idea is to compare the induced suffix/LCP items with their corresponding items in sa/lcp during the inducing process. Specifically, when a suffix/LCP item v_1 is induced into a bucket, we check if it is equal to the corresponding item v_2 in sa/lcp. If $v_1 = v_2$, then v_2 is correct and we further use this value to induce the remaining suffix/LCP items. The key point is to quickly retrieve the items of sa/lcp in external memory. This can be done by conducting sequential I/O operations if we provide a read

1. An overview of the induction phase is given in Appendix 6.

TABLE 1
Corpus, n in Gi, 1 Byte per Character

Corpora	$\|\Sigma\|$	n	Description
enwiki	256	74.7	An XML dump of English Wikipedia, available at https://dumps.wikimedia.org/enwiki, dated as 16/05/01.
uniprot	96	2.5	UniProt Knowledgebase, available at ftp://ftp.expasy.org/databases/uniprot/current_release/knowledgebase/complete/, dated as 16/05/11.
proteins	27	1.1	Swissprot database, available at http://pizzachili.dcc.uchile.cl/texts/protein, dated as 06/12/15.

pointer together with a buffer for each suffix/LCP subbucket. We describe below more details of the modified inducing process, where lp_1/lp_2 and sp_1/sp_2 indicate the next items to be visited in the L-type and S-type subbuckets, respectively.

(a) Let $lp_1[c]$ and $lp_2[c]$ point to the leftmost items of $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{lcp\_bkt_L}(c)$, for $c \in [0, \Sigma)$.

(b) Scan sa and lcp rightward to induce the L-type suffixes and their LCP-values. For each induced suffix p (with a heading character c_0) and its LCP-value q: (1) check if $p = lp_1[c_0]$ and $q = lp_2[c_0]$; (2) move $lp_1[c_0]$ and $lp_2[c_0]$ to the next items on the right.

(a) Let $sp_1[c]$ and $sp_2[c]$ point to the rightmost items of $\mathsf{sa\_bkt_S}(c)$ and $\mathsf{lcp\_bkt_S}(c)$, for $c \in [0, \Sigma)$.

(b) Scan sa and lcp leftward to induce the S-type suffixes and their LCP-values. For each induced suffix p (with a heading character c_0) and its LCP-value q: (1) check if $p = sp_1[c_0]$ and $q = sp_2[c_0]$; (2) move $sp_1[c_0]$ and $sp_2[c_0]$ to the next items on the left.

3.4 Analysis

Algorithm 2 mainly consists of two steps, where the first step for checking sa^* and lcp^* can be done within sorting complexity and the second step for checking sa and lcp can also be done in sorting complexity for using external-memory sorters and priority queues. In our current program for this algorithm, the peak disk use is reached in the second step, specifically, when computing the BWT from sa and x for use in the inducing process.

4 Experiments

4.1 Setup

For implementation simplicity, our programs for the algorithms proposed in the previous sections use the external-memory containers provided by the STXXL library [29] to manage read/write operations on disks. We make a performance evaluation by running them on the real-world corpora listed in Table 1, where three measures normalized by the size of input string are investigated:

- RT: Running time, in microseconds.
- PDU: Peak disk use of external memory, in bytes.
- IOV: Amount of data read from and write to external memory, in bytes, where each integer is 40-bit.

The experimental platform is a server with an Intel Core i3-550 CPU, 4 GiB RAM and 2 TiB HD. All the programs are compiled by gcc/g++ 4.8.4 with -O3 options on ubuntu 14.04 64-bit operating system and each program is allowed to use 3 GiB RAM. For simplicity, "ProgA" and "ProgB" represent the programs for Algorithms 1 and 2, respectively.

Algorithm 2. The Algorithm Based on Corollary 2

```
Function CheckByIS($x$, $sa$, $lcp$, $n$)
    $ST_1 := [(sa[i], i, null) | i \in [0, n)]$
    sort the tuples in $ST_1$ by the 1st component;
    $pos := -1$
    for $i \in (n, 0]$ do
        $e := ST_1$.top(), $ST_1$.pop()
        if $x[i]$ is S*-type then
            if $pos \geq e.1st$ then
                return false        // condition 1 is violated
            end
            $ST_2$.push($e$), $pos := e.1st$
        end
    end
    sort the tuples in $ST_2$ by the 2nd component;
    $i := 0, j := 0, lcp_{min} := max\_val$
    while $ST_2$.NotEmpty() do
        $e := ST_2$.top(), $ST_2$.pop()
        while true do
            $lcp_{min} := \min(lcp_{min}, lcp[i])$
            if $e.2nd = i$ then
                $sa^*[j] := e.1st$, $lcp^*[j] := lcp_{min}$, $j := j + 1$,
                $i := i + 1$
                break
            end
            $i := i + 1$
        end
        $lcp_{min} := max\_val$
    end
    if CheckByFP($x, sa^*, lcp^*, n_1$) $= false$ then
        return false        // conditions 2 or 3 is violated
    end
    $(sa', lcp')$ := InducingProcess($x, sa^*, lcp^*$)
    for $i \in [0, n)$ do
        if $sa[i] \neq sa'[i] \,\|\, lcp[i] \neq lcp'[i]$ then
            return false        // condition 4 is violated
        end
    end
    return true
```

4.2 Results

Fig. 2 illustrates the performance comparison of ProgA and ProgB on different datasets, where "enwiki_8g" consists of the leftmost 8 GiB extracted from "enwiki". As depicted, ProgB runs slower than ProgA by around 20 percent. The speed gap is mainly due to the difference in I/O performance. Specifically, the I/O volume of ProgA keeps at 155 n for all the three datasets, while that of ProgB rises up to nearly 200 n on average. Besides, the peak disk use of ProgB is about $26/40 = 0.65$ as ProgA. Recall that Algorithm 2

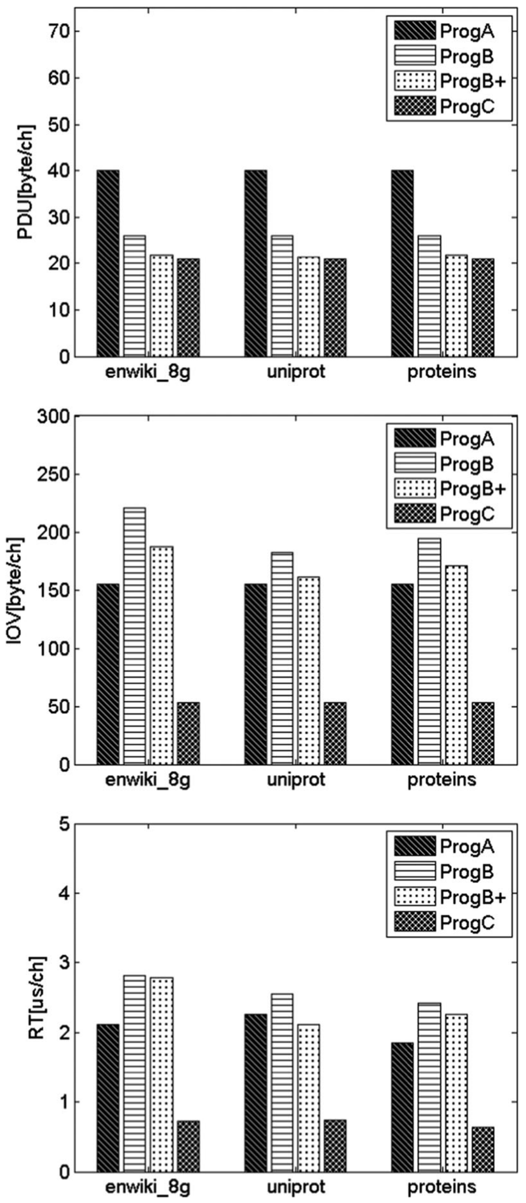

Fig. 2. Performance of ProgA, ProgB, ProgB+ and ProgC for different corpora.

invokes Algorithm 1 to check the suffix and LCP arrays for the S*-type suffixes. Because at most one out of every two successive characters in the input string is S*-type, the consumption for checking the suffix and LCP arrays of S*-type suffixes in ProgB is expected to be half as that for checking the given arrays in ProgA. For a better insight, we collect in Table 2 the performance overhead of ProgB and ProgA when checking the suffix and LCP arrays of S*-type suffixes and all, respectively. As can be observed, the mean ratio of the number of S*-type suffixes to the number of all the suffixes is around 0.30 for the datasets under investigation, while the mean ratios of time, space and I/O volume for checking sa^* and lcp^* to that for checking sa and lcp are 0.38, 0.57 and 0.60, respectively.

The above observations indicate that ProgB reaches its peak disk use when checking the final suffix and LCP arrays during the inducing process, i.e., the inducing process constitutes the performance bottleneck of the whole algorithm. By adopting the space optimization scheme introduced in Section 3.3, we adapt Algorithm 2 and evaluate the tuned version of ProgB, called ProgB+, in comparison with ProgA and ProgB. Fig. 2 shows that the maximum space requirement for ProgB+ is about $21n$, which is much less than that of ProgB and even only half as that of ProgA. In addition, progB+ outperforms its prototype with respect to time and I/O efficiency and is faster than ProgA when handling ”uniprot”. We also investigate the performance trend of the three programs on the prefix of ”enwiki” with the length varying in $\{1, 2, 4, 8\}$ GiB. In Fig. 3, the peak disk use for each program remains unchanged, but their speed become slower as the prefix length increases due to the performance degradation of the external memory sorter used in our programs. This can be also observed from Table 2, where ProgB+ keeps the I/O volume around 90 n with the prefix length of “enwiki” varying from 1 to 8 GiB but its running time rises from 1.05 to 1.33.

Because there is no solution for checking both suffix and LCP arrays in the existing literature, comparing the outputs of two builders could be a way for checking (even though it is not checking in the strict sense, for both outputs may be incorrect). In the next experiment, we compare our programs with two builders for suffix and LCP arrays as follows, where each of them combines an existing suffix sorter with an LCP builder:

- Solution 1: Use eSAIS for building SA and the sequential version of Sparse-ϕ [23] for building LCP array.
- Solution 2: Use pSAscan [22] for building SA and the parallel version of Sparse-ϕ for building LCP array.

We select these programs because they are currently the fastest suffix and LCP arrays builders available to us. A runtime breakdown of the programs for these solutions on the prefixes of ”enwiki” is given in Table 3. The program for

TABLE 2
A Performance Comparison of Checking the Suffix and LCP Arrays of S*-Type Suffixes to Checking That of All the Suffixes

Dataset	# of suffixes			PDU			IOV			RT		
	S*-type	all	ratio	S*-type	all	ratio	S*-type	all	ratio	S*-type	all	ratio
enwiki_1g	329,810,376	1,073,741,824	0.31	15.67	40	0.39	89.94	155	0.58	1.05	1.70	0.62
enwiki_2g	650,901,939	2,147,483,648	0.30	15.41	40	0.39	89.18	155	0.58	1.22	1.85	0.66
enwiki_4g	1,301,327,878	4,294,967,296	0.30	15.45	40	0.39	89.14	155	0.58	1.19	1.89	0.63
enwiki_8g	2,586,471,839	8,589,934,592	0.30	15.35	40	0.38	88.80	155	0.57	1.33	2.14	0.62
uniprot	829,262,945	3,028,811,776	0.27	13.94	40	0.35	83.80	155	0.54	1.04	2.26	0.46
proteins	379,092,002	1,184,366,592	0.32	16.21	40	0.41	92.29	155	0.60	1.14	1.85	0.62
mean	1,012,811,163	3,189,156,522	0.30	15.34	40	0.38	88.86	155	0.57	1.16	1.95	0.60

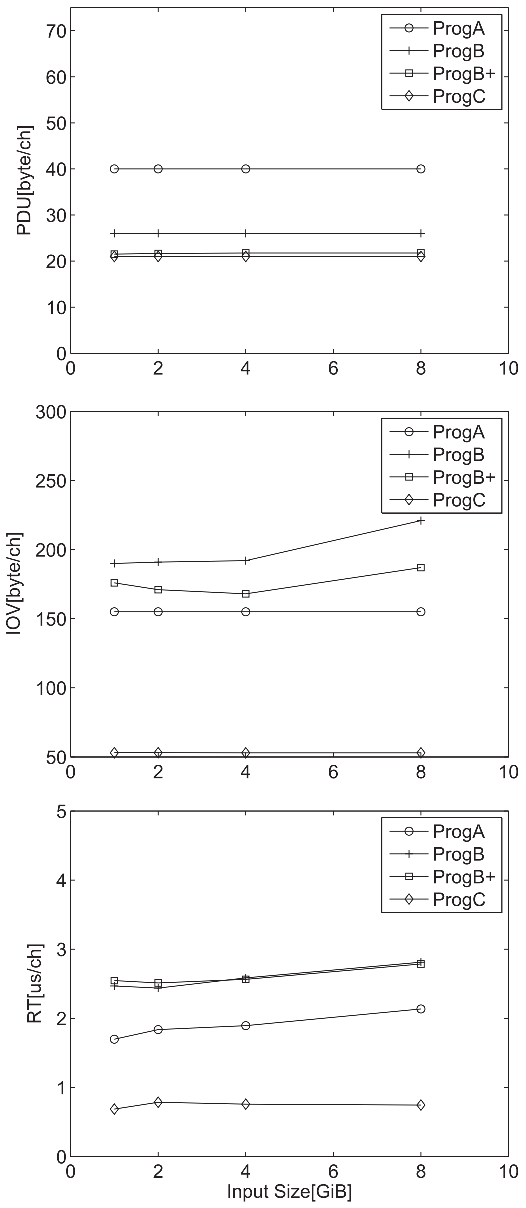

Fig. 3. Performance of ProgA, ProgB, ProgB+ and ProgC for prefixes of ”enwiki”.

Solution 2 is about two times faster than that for Solution 1 and twice as fast as ProgA, which is mainly due to the high speed of pSAscan in this experiment. However, it is worthy of pointing out that both pSAscan and Sparse-ϕ are of the time and I/O complexities proportional to n^2/M. This is much higher than eSAIS and our checking algorithms when n increases, and thus poses a strict limitation to the scalability of Solution 2. As reported in [22], when n is considerably greater than M, eSAIS is much more time and I/O efficient than pSAscan. In this experiment, pSAscan builds the SA for “enwiki_8g” in time double as that for “enwiki_1g”. For big n, it is reasonable to compare the results of our programs with that of Solution 1.

Currently, the method described in [26] is the only known in the literature for checking SA. Despite that it can check SA only, in order to give a rough image for the performance of our methods, we implement it by STXXL and compare our implementation with ProgA and ProgB/ProgB+ in Figs. 2 and 3. This method and its program are denoted by ”Method C” and ”ProgC”, respectively. ProgC is about two times faster than ProgA and three times faster than ProgB+, where the runtime is consistent with the I/O volume. This performance gap can be significantly narrowed by improving the algorithm and program designs of Methods A and B, using the techniques discussed below.

4.3 Discussion

4.3.1 Improvement in Algorithm Design

There are several ways to improve the algorithm designs of the proposed checking methods. First, Algorithm 1 sorts the tuples of $ST_1/ST'_1, ST_2/ST'_2, ST_3/ST'_3$ in sequence. These sorting processes are independent and can be parallelized on computation platforms with multiple disks supporting parallel reads and writes. Second, Method B verifies the suffix and LCP arrays using the induced sorting principle. At the time of writing this paper, the existing IS algorithms are naturally sequential. Recently, we have been conducting a study to design parallel IS algorithms, this work will also help improve the design of Algorithm 2. Third, both methods A and B assume a constant or integer alphabet in this paper. However, in practice, an input string is commonly of a constant alphabet, e.g., 4 and 256 characters for genome and text, respectively. In this case, Method B can be improved for better time and space performance by inducing the final suffix and LCP arrays directly from sa_S/lcp_S or sa_L/lcp_L, which consist of all the sorted S-type or L-type suffixes with their LCP-values and can be obtained as follows. Given the alphabet is constant, we first scan the input string once to get the statistics for buckets in the input suffix array. Without loss of generality, suppose that the S-type characters are less, we scan the suffix array once to get sa_S/lcp_S by using the bucket statistics to on-the-fly determine a scanned suffix is S-type or not. Then we check sa_S/lcp_S by using Algorithm 1 and induce sa'/lcp' from them. In this way, we avoid the two integer sorts in the current fashion of Method B for retrieving sa^*/lcp^* and speed up the inducing process by nearly half as well.

TABLE 3
A Runtime Comparison for the Programs of Two Construction Solutions and Ours

Dataset	Solution 1			Solution 2			ProgA	ProgB+
	eSAIS	sequential sparse-ϕ	total	pSAscan	parallel sparse-ϕ	total		
enwiki_1g	2.21	0.61	2.82	0.39	0.59	0.98	1.70	2.54
enwiki_2g	2.63	0.53	3.16	0.47	0.53	1.00	1.84	2.51
enwiki_4g	2.90	0.63	3.53	0.59	0.40	0.99	1.89	2.56
enwiki_8g	3.02	0.63	3.65	0.83	0.45	1.28	2.13	2.79

4.3.2 *Improvement in Program Design*

Our programs are coded for experimental study only, from engineering aspects, there is still a big margin for better implementation. For example, both ProgA and ProgB/ProgB+ consume long CPU time and large I/O volume for sorting data in external memory. We currently use the containers provided by STXXL to execute the sorting processes without designing a specific sorter optimized for our purpose. It is possible to speed up each sorting process by high-performance radix-sort GPU algorithms. Second, all our programs require a disk space significant more than what it is really needed. The main reason is the disk space for saving temporary data is not freed in time even if the data is not needed any more. This is an implementation issue due to STXXL in use, can be solved by storing the temporary data using multiple files and deleting each file immediately when it is obsolete.

Several algorithms for induced sorting a suffix and/or LCP arrays were proposed these recent years [10], [12], [13], with different methods for solving the key problem of retrieving the preceding character of a sorted suffix in the inducing process. A recent work [30] engineering these induced sorting methods with some implementation optimizing techniques achieves a significant improvement over the previous results. As reported, the peak disk use is around $8n$ for 40-bit integers. Because the induced sorting process is the performance bottleneck for Method B, it is reasonable to expect that a better engineering implementation of the method will yield a remarkable improvement in both time and space. An optimized engineering of our methods is out of the scope of this paper and will be addressed elsewhere.

4.3.3 *Miscellaneous*

Method A is more general than the other alternatives, it can be generalized to check the correctness of the lexical order and LCP values of any pairs of suffixes, this makes it possible for verifying any full or sparse suffix/LCP array of any order. On the other hand, Method B can only check suffix and LCP arrays, while Method C is specific for checking SA only. This feature of Method A makes it applicable to various scenarios. For example, a suffix/LCP array may be broken due to software or hardware malfunctions. If a backup is not available and it is time-consuming to rebuild the whole array, then we can locate the bad areas using Algorithm 1 and restore the partial SA for each area by calling a sparse SA construction algorithm. Another example is to check the correctness of a sparse SA, in this case, the number of suffixes in a sparse SA is commonly much smaller than that in the full SA, Algorithm 1 could be an efficient verification solution.

5 Conclusions

Two methods are proposed here for probabilistically checking a pair of given suffix and LCP arrays. Theoretically, the external-memory algorithms designed by these methods have better time and I/O complexities compared to the existing fastest construction algorithms. Our experimental results indicate that the current programs for Algorithm 2 designed by Method B run slower than that for Algorithm 1 designed by Method A, but they are much more space efficient than the latter. We also show in Section 4 that there still remains much room for improving the algorithm and program designs of the proposed methods. Particularly, our experimental program for Method A can be parallelized to achieve higher time performance, while that for method B can be further optimized for checking arrays of constant alphabets that are most common in practice.

The IS method has been applied to successfully design a number of suffix and LCP arrays construction algorithms. A recent work [30] reports that a careful engineering of the IS in external memory can build a suffix array using around 8 n bytes for $n \leq 2^{40}$, which is approaching 6 n bytes for the IS in internal memory. Besides, it runs the fastest for large n in the experiments therein. This convinces that the IS method could be a stand for developing potentially optimal solutions for building suffix/LCP arrays. We design here the algorithms for checking a pair of given suffix and LCP arrays. In another paper, we will come up with a solution for building and checking a suffix/LCP array simultaneously using the IS method. By this way, no additional checker is needed to be distributed with a suffix/LCP array builder using the IS method.

Acknowledgments

The work of G. Nong was supported by the Guangzhou Science and Technology Program grant 201707010165 and the Project of DEGP grant 2014KTSCX007. The work of W. H. Chan was supported by GRF (810012), Research Grant Council, Hong Kong SAR.

References

[1] M. Abouelhodaa, S. Kurtzb, and E. Ohlebuscha, “Replacing suffix trees with enhanced suffix arrays,” *J. Discr. Algorithms*, vol. 2, no. 1, pp. 53–86, Nov. 2004.

[2] U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[3] J. Kärkkäinen and P. Sanders, “Simple linear work suffix array construction,” in *Proc. 30th Int. Colloq. Autom. Languages Program.*, 2003, pp. 943–955.

[4] P. Ko and S. Aluru, “Space efficient linear time construction of suffix arrays,” in *Proc. 14th Annu. Symp. Combinatorial Pattern Matching*, 2003, pp. 200–210.

[5] D. K. Kim, J. S. Sim, H. Park, and K. Park, “Linear time construction of suffix arrays,” in *Proc. 14th Annu. Symp. Combinatorial Pattern Matching*, 2003, pp. 186–199.

[6] G. Nong, S. Zhang, and W. H. Chan, “Two efficient algorithms for linear time suffix array construction,” *IEEE Trans. Comput.*, vol. 60, no. 10, pp. 1471–1484, Oct. 2011.

[7] R. Dementiev, J. Kärkäinen, J. Mehnert, and P. Sanders, “Better external memory suffix array construction,” *ACM J. Exp. Algorithmics*, vol. 12, no. 3, pp. 4:1–4:24, Aug. 2008.

[8] P. Ferragina, T. Gagie, and G. Manzini, “Lightweight data indexing and compression in external memory,” *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.

[9] G. Manzini and P. Ferragina, “Engineering a lightweight suffix array construction algorithm,” *Algorithmica*, vol. 40, pp. 33–50, Sep. 2004.

[10] T. Bingmann, J. Fischer, and V. Osipov, “Inducing suffix and LCP arrays in external memory,” in *Proc. 15th Workshop Algorithm Eng. Exp.*, 2012, pp. 88–102.

[11] J. Kärkkäinen and D. Kempa, “Engineering a lightweight external memory suffix array construction algorithm,” in *Proc. 2nd Int. Conf. Algorithms Big Data*, 2014, pp. 53–60.

[12] G. Nong, W. H. Chan, S. Zhang, and X. F. Guan, "Suffix array construction in external memory using D-critical substrings," *ACM Trans. Inform. Syst.*, vol. 32, no. 1, pp. 1:1–1:15, Jan. 2014.

[13] G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu, "Induced sorting suffixes in external memory," *ACM Trans. Inform. Syst.*, vol. 33, no. 3, pp. 12:1–12:15, Mar. 2015.

[14] J. Fischer, "Inducing the LCP-array," in *Algorithms Data Struct.*, vol. 6844, pp. 374–385, 2011.

[15] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–10.

[16] T. K. G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. 12th Annu. Symp. Combinatorial Pattern Matching*, 2001, pp. 181–192.

[17] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Proc. 20th Annu. Symp. Combinatorial Pattern Matching*, 2009, pp. 181–192.

[18] S. J. Puglisi and T. Andrew, "Space-time tradeoffs for longest-common-prefix array computation," in *Proc. 19th Int. Symp. Algorithms Comput.*, 2008, pp. 124–135.

[19] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the GPU," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 197–206.

[20] V. Osipov, "Parallel suffix array construction for shared memory architectures," in *Proc. Int. Symp. String Process. Inform. Retrieval*, 2012, pp. 379–384.

[21] L. Wang, S. Baxter, and J. Owens, "Fast parallel suffix array on the GPU," in *Proc. 21st Int. Conf. Parallel Distrib. Comput.*, 2015, pp. 573–587.

[22] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel external memory suffix sorting," in *Proc. 26th Annu. Symp. Combinatorial Pattern Matching*, 2015, pp. 329–342.

[23] J. Kärkkäinen and D. Kempa, "Faster external memory LCP array construction," in *proc. 24th European Symp. Algorithms*, Aug. 2016, pp. 61:1–61:16.

[24] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better external memory suffix array construction," *ACM J. Exp. Algorithmics*, vol. 12, pp. 3.4:1–3.4:24, Aug. 2008.

[25] P. Bille, et al., "Sparse suffix tree construction in small space," in *Proc. Int. Colloquium Autom. Languages Program.*, 2013, pp. 148–159.

[26] S. Burkhardt and J. Kärkkäinen, "Fast lightweight suffix array construction and checking," in *Proc. 14th Symp. Combinatorial Pattern Matching*, 2003, pp. 55–69.

[27] R. Karp and M. Rabin, "Efficient randomized pattern matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, Mar. 1987.

[28] L. Arge and M. Thorup, "RAM-efficient external memory sorting," *Algorithms Comput.*, vol. 9293, no. 3, pp. 491–501, 2013.

[29] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard template library for XXL data sets," *Software: Practice Exp.*, vol. 38, no. 6, pp. 589–637, 2008.

[30] J. Kärkkäinen, D. Kempa, S. J. Puglisi, and B. Zhukova, "Engineering external memory induced suffix sorting," in *Proc. 19th Workshop Algorithm Eng. Exp.*, 2017, pp. 98–108.

Yi Wu received the BSc and MSc degrees in computer science from the Central-South University, in 2009 and the Sun Yat-sen University in 2012, both in China, respectively. He is currently working towards the PhD degree in computer science at the Sun Yat-sen University, China. His current research interests are switching and routing, and string processing for massive data.

Ge Nong received the BE and ME degrees, both in computer engineering, from the NanJing University of Aeronautics and Astronautics, in 1992 and the South China University of Science and Technology, in 1995, respectively. He received the PhD degree in computer science from the Hong Kong University of Science and Technology, in 1999. Then, he joined STMicroelectronics as a senior researcher with R&D on IC and system technologies for high-speed switches and routers. He is now a professor in the Department of Computer Science, Sun Yat-sen University in Guangzhou, China. His current research interests include algorithms, computer and communication networks, switching theory and performance evaluation. He is a senior member of the IEEE.

Wai Hong Chan received the BSc, MPhil and PhD degrees in mathematical science from the Hong Kong Baptist University, Hong Kong Special Administrative Region, China, in 1994, 1996 and 2003 respectively. He joined the Department of Mathematics and Information Technology, the Education University of Hong Kong in 2011, and now the head of Department. His current research interests include algorithm design, quantum information, graph theory and combinatorics.

Ling Bo Han received the BS degree in computer science from the Taiyuan Normal University, in 2006 and the MS degree in computer science from the Guangxi Normal University in 2009, both in China. He is currently working toward the PhD degree in the School of Data and Computer Science, the Sun Yat-sen University, China. His current research interests are data mining, and string processing for massive data.