Checking Big Suffix and LCP Arrays by Probabilistic Methods

Yi Wu, Ge Nong, Wai Hong Chan, Ling Bo Han

Abstract—For full-text indexing of massive data, the suffix and LCP (longest common prefix) arrays have been recognized as the fundamental data structures and should be verified to ensure their correctness after construction. We propose in this paper two methods to probablisitically check the suffix and LCP arrays in external memory by using a Karp-Rabin fingerprinting function, in terms of that the checking result is wrong with only a negligible probability. The first method checks the lexical order and the LCP-value of two neighboring suffixes in the given suffix array by computing and comparing the fingerprints of their LCPs. The second method first applies the fingerprinting technique to check a subset of the given suffix and LCP arrays, then it produces the whole arrays from the verified parts following the induced-sorting principle and compares the induced and input copies with each other for verification.

Index Terms—Suffix and LCP arrays, verification, Karp-Rabin fingerprinting function.

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 dozens of applications, these two data structures are combined to constitute a powerful full-text index for massive data, called enhanced suffix array [1], which is more space efficient than suffix tree and applicable to emulating any searching functionalities provided by the latter in the same time complexity. During the past decades, much effort has been put on the development of designing efficient suffix array construction algorithms (SACAs). Specifically, the first internal-memory algorithm for building SA was introduced in [2]. From then on, a plethora of SACAs have been proposed on different computation models, e.g., internal memory [3], [4], [5], [6], external memory [7], [8], [9], [10], [11], [12], [13] and shared memory models [14], [15], [16], [17]. In respect of the design on LCP-array construction algorithms (LACAs), the existing works can be classified into two categories, where the algorithms of the first category compute the suffix and LCP arrays in the same time [10], [18], [19] and that of the second category take the suffix array (SA) and/or Burrows-Wheeler transform (BWT) as input to facilitate the computation [15], [18], [20], [21], [22], [22], [23].

While the study for efficient construction of suffix and LCP arrays is evolving, the programs implementing the proposed algorithms are commonly provided "as is", with the purpose only for the performance evaluation experiemnts of the articles where they are reported. That is, these programms give no guarantee that they have correctly implemented the proposed algorithms. The programs for recently proprosed algorithms are becoming much more complicated than before, casuing more difficulties for program verifying and debugging[1]. As a common practice, a suffix or LCP array checker is also provided for verifying the correctness of a constructed array. For example, such a checker is provided in the software packages eSAIS [**?**] (DC3, pScan and etc? more and better) for constructing suffix and/or LCP arrays. In addition to help avoid implementation bugs, a checker is also demanded for an array constructed by a probabilistic algorithm [24]. In this case, the array is correct with a probability and hence must be verified by a checker to ensure its correctness.

As far as we know, the work presented in [25] is the only SA checking method that can be found in the existing literature, and no efficient approach for the LCP-array verification has been reported yet. In particular, 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 external memory algorithms for checking the suffix and LCP arrays of massive data.

1.2 Contribution

Our contribution includes two checking methods for the given suffix and LCP arrays in external memory.

The main idea of the first method is to test the lexical order and the LCP-value of two neighboring suffixes in a suffix array by literally comparing their characters. To reduce time complexity for a comparison between two sequences of characters, a Karp-Rabin fingerprinting function is employed to transform each sequence into a single integer, called fingerprint, such that the equality of two sequences can be correctly checked with a negiligible error probability by comparing their fingerprints in constant time.

- *Y. Wu, G. Nong (corresponding author) and L. B. Han are with the Department of Computer Science, Sun Yat-sen University, Guangzhou 510275, China. E-mails: wu.yi.christian@gmail.com, issng@mail.sysu.edu.cn, hanlb@mail2.sysu.edu.cn.*
- *Wai Hong Chan (corresponding author) is with the Department of Mathematics and Information Technology, The Education University of Hong Kong, Hong Kong. E-mail: waihchan@ied.edu.hk.*

1. In our studies before, we have exprienced problems caused by bugs of the exisitng programs.

By using the same fingerprinting technique, the second method first verifies a subset chosen from the input arrays and then produces a copy of the suffix and LCP arrays from the verified subset following the induced sorting (IS) principle. Given that the inducing process is correct, the input arrays are considered to be right with a high probability if they are equal to the induced copies.

The remainder of this paper is organized as follows. We first describe the proposed two checking methods in Sections 2 and 3, then present the experimental results in Section 4,and give the conclusion in Section 5.

2 Method A

2.1 Preliminaries

Given an input string $x[0,n)$ drawn from an alphabet Σ, the suffix array of x, denoted by sa, is a permutation of $\{0,1,...,n-1\}$ such that $\mathsf{suf}(sa[i]) < \mathsf{suf}(sa[j])$ is satisfied for $0 \leq i < j < n$, 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 $\mathsf{suf}(sa[i-1])$ and $\mathsf{suf}(sa[i+1])$ are the lexical neighbors of $\mathsf{suf}(sa[i])$ in sa. 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 $i \in [1,n)$.

2.2 Idea

The lexical order and the LCP-value of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ can be determined by literally comparing their characters from left to right. Because all the suffixes differ in length and end with a common character, there must exist $k \in [0,n)$ such that $x[i,i+k) = x[j,j+k)$ and $x[i+k] \neq x[j+k]$. According to Lemma 1, this method can be also applied to checking suffix and LCP arrays, but it suffers from high time complexity as the two substrings indicated by the LCP-value for each pair of neighboring suffixes in sa take at worst $\mathcal{O}(n)$ character-wise comparisons.

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 $\{0,1,\ldots,n-1\}$.
(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 seen from the definition of suffix and LCP arrays. Specifically, condition (1) demonstrates that all the suffixes in x are sorted in sa, while conditions (2)-(3) indicate that the lexical order and the LCP-value of any two neighboring suffixes in sa are both correct. □

An alternative is to exploit a perfect hash function (PHF) 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 equality of two substrings can be determined by comparing the corresponding hash values instead. The key point here is how to efficiently compute the hash values of $x[sa[i], sa[i]+lcp[i]-1]$ and $x[sa[i-1], sa[i-1]+lcp[i]-1]$ for all $i \in [1,n)$. Taking into account the high cost of finding a PHF to meet this requirement, we prefer using a Karp-Rabin fingerprinting function [26] to transform a substring into its integer form, called fingerprint. Specifically, suppose L is a prime and δ is randomly chosen from $[1,L)$, the fingerprint $\mathsf{fp}(i,j)$ of a substring $x[i,j]$ can be calculated by using Formulas 1- 3 as following: scan x rightward to iteratively compute $\mathsf{fp}(0,k)$ for all $k \in [0,n)$ according to Formulas 1-2, meanwhile, record $\mathsf{fp}(0,i-1)$ and $\mathsf{fp}(0,j)$ and subtract the former from the latter to obtain $\mathsf{fp}(i,j)$ according to Formula 3.

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

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

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

It is worthy of mentioning that two equal substrings always share a common fingerprint, but the inverse is not true. Fortunately, it has been proved in [26] that the probability of a false match can be reduced to a negligible level by setting L to a large value[2]. This leads us to the following conclusion.

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

2.3 Algorithm

Section 2.2 indicates that we can perform verification for the given suffix and LCP arrays by testing the conditions of Corollary 1. Based on this idea, we introduce below a linear algorithm for checking sa and lcp on random access models.

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

S2 Scan sa and lcp rightward with i increasing from 1 to $n-1$. For each scanned $sa[i]$ and $lcp[i]$, let $u = sa[i], v = lcp[i], w = sa[i-1]$ and performs substeps (a)-(c) sequentially:
- (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 $x[u+v] > x[w+v]$.

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

Two zero-initialized arrays fp and mk are employed to faciliate the checking process, where the former is for storing the fingerprints of all the prefixes in x and the latter is for checking the existence of $\{0,1,...,n-1\}$ in sa. Clearly, this algorithm consumes $\mathcal{O}(n)$ time and space when running in internal memory. However, if the input can not be wholly accommodated into RAM, it may suffer from a performance degradation due to frequent random I/O operations for reading elements of x, sa and lcp from external memory during the execution of S2.

We propose Algorithm 1 to perform the checking process in an I/O friendly way, which conducts external-memory sorts to avoid random accesses to external disks. At the

2. This property is utilized in [24] to design a probabilistic algorithm for computing a sparse suffix array.

Algorithm 1: The Algorithm for checking the conditions of Corollary 1.

```
Function CheckByFP(x, sa, lcp, n)
    $ST_1 := [(sa[i], i, null) | i \in [0, n)]$.
    $ST_2 := [(sa[i] + lcp[i+1], i, null, null) | i \in [0, n-1)]$.
    $ST_3 := [(sa[i] + lcp[i], i, null, null) | i \in [1, n)]$.
    sort tuples in $ST_1$, $ST_2$ and $ST_3$ by 1st component.
    $fp := 0$
    for $i \in [0, n]$ do
        if $ST_1$.notEmpty() and $ST_1$.top().$1st = i$ then
            $e := ST_1$.top(), $ST_1$.pop(), $e.3rd := fp$, $ST'_1$.push($e$)
        end
        else
            return false                          // condition (1) is violated
        end
        while $ST_2$.notEmpty() and $ST_2$.top().$1st = i$ do
            $e := ST_2$.top(), $ST_2$.pop(), $e.3rd := fp$, $e.4th := x[i]$, $ST'_2$.push($e$)
        end
        while $ST_3$.notEmpty() and $ST_3$.top().$1st = i$ do
            $e := ST_3$.top(), $ST_3$.pop(), $e.3rd := fp$, $e.4th := x[i]$, $ST'_3$.push($e$)
        end
        $fp := fp \cdot \delta + x[i] \mod P$
    end
    sort tuples in $ST_1$, $ST_2$ and $ST_3$ by 2nd component.
    for $i \in [1, n-1)$ do
        $fp_1 := ST'_1$.top().$3rd$, $ST'_1$.pop(), $fp_2 := ST'_2$.top().$3rd$, $ch_1 := ST'_2$.top().$4th$, $ST'_2$.pop()
        $\hat{fp_1} = fp2 - fp1 \cdot \delta^{lcp[i]} \mod P$
        $fp_1 := ST'_1$.top().$3rd$, $fp_3 := ST'_2$.top().$3rd$, $ch_2 := ST'_3$.top().$4th$, $ST'_3$.pop()
        $\hat{fp_2} = fp3 - fp1 \cdot \delta^{lcp[i]} \mod P$
        if $\hat{fp_1} \neq \hat{fp_2}$ or $ch_1 \leq ch_2$ then
            return false                          // condition (2) or (3) is violated
        end
    end
    return true
```

very beginning, the algorithm first scans sa and lcp to produce ST_1, ST_2, ST_3 and sorts the tuples of them by 1st component in ascending order (lines 2-5). Then, it computes the fingerprints of all the prefixes according to Formulas 1-2 and assign them to the sorted tuples in lines 6-21 as following: when finished computing $\mathsf{fp}(0, i-1)$, extract each tuple e with $e.1st = i$ from ST_1, ST_2, ST_3 and update them with $\mathsf{fp}(0, i-1)$ and $x[i]$ (if required), where the tuples are forwarded to ST'_1, ST'_2, ST'_3 after updating and sorted back to their original order (line 22). During the process, we determine whether or not the 1st components of all the tuples in ST_1 consistute a permutation of $\{0, 1, ..., n-1\}$ to test the first condition of Corollary 1 (lines 9-14). Finally, it repeatedly retrieves the top tuples from ST'_1, ST'_2, ST'_3 and applies Formulas 3 to compute the fingerprints of two substrings specified by their 1st components for ensuring the satisfaction of conditions (2)-(3). A point to be explained here is how to compute $\delta^{lcp[i]}$ in lines 25 and 27. Let $e := lcp[i]$, our method first decomposes e into $\Sigma_{i=0}^{\lceil \log 2^n \rceil} k_i \cdot 2^i$ and then computes $\Pi_{i=0}^{\lceil \log 2^n \rceil} \delta^{k_i \cdot 2^i}$ to obtain δ^e, where $k_i \in \{0, 1\}$. Following this way, the answer can be returned in $\mathcal{O}(\lceil \log 2^n \rceil)$ time using $\mathcal{O}(\lceil \log 2^n \rceil)$ space for storing $\{\delta^1, \delta^2, \ldots, \delta^{2^{\lceil \log 2^n \rceil}}\}$.

2.4 Analysis

Algorithm 1 performs multiple scans and sorts for arrays of n fixed-size tuples using external memory. Consider an external memory model with RAM size M, disk size D and block size B, all are in words, then the time and I/O complexities for a scan are $\mathcal{O}(n)$ and $\mathcal{O}(n/B)$, respectively, while those for a sort with an integer key are $\mathcal{O}(n \log_{M/B}(n/B))$ and $\mathcal{O}((n/B) \log_{M/B}(n/B))$, respectively [27]. Besides, the algorithm reaches its peak disk use when sorting tuples in lines 5 and 22. An optimization for reducing maximum space requirements is to compute the fingeprints indicated by ST_1, ST_2, ST_3 separately. This will lead to a small increase in total I/O volume as it needs to compute $\{\mathsf{fp}(0,0), \mathsf{fp}(0,1), ..., \mathsf{fp}(0, n-1)\}$ two more times.

2.5 Example

We demonstrate in this part an example for better understanding. xxx.

3 Method B

Our experimental study in Section 4 shows that Algorithm 1 is quite space consuming, its peak disk use is 40 bytes per input character. In this section, we describe an alternative based on the induced-sorting principle. Compared with Algorithm 1, the algorithm designed by this method only takes half space on real-world datasets.

3.1 Preliminaries

Before our presentation, we first introduce some notations for description convenience.

Character and suffix classification. All the characters in x are classified into three types, namely L-, S- and S*-type. Detailedly, $x[i]$ 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 S- and L-type respectively, then $x[i]$ is also an S*-type character. Moreover, the type of a suffix is the same as that of its heading character.

Suffix and LCP buckets. Suppose sa is correct, then suffixes in sa are naturally partitioned into multiple buckets and those with an identical heading character are grouped into one bucket occupying a contiguous interval. Further, a bucket can be divided into two parts, where the left and right part contain L- and S-type suffixes, respectively. For short, we use $\mathsf{sa\_bkt}(c)$ to denote the bucket storing suffixes starting with c and $\mathsf{sa\_bkt_L}(c)/\mathsf{sa\_bkt_S}(c)$ to denote its left/right sub-bucket. Accordingly, lcp can be decomposed into multiple buckets as well, where $\mathsf{lcp\_bkt}(c)/\mathsf{lcp\_bkt_L}(c)/\mathsf{lcp\_bkt_S}(c)$ store the LCP-values of suffixes in $\mathsf{sa\_bkt}(c)/\mathsf{sa\_bkt_L}(c)/\mathsf{sa\_bkt_S}(c)$ and their left lexical neighbors.

Suffix and LCP arrays for S-type suffixes.* Suppose the number of S*-type suffixes in x is n_1, $sa^*[0, n_1)$ and $lcp^*[0, n_1)$ indicate the lexical order and the LCP-values of these S*-type suffixes, where $x[sa^*[i]]$ is the heading character of the $(i+1)$-th smallest S*-type suffix.

Type array. The type array t records the type of $x[i]$ in $t[i]$ for $i \in [0, n)$.

3.2 Idea

The induced sorting principle has been employed to design algorithms for constructing suffix and LCP arrays in both internal and external memory. These algorithms mainly consist of a reduction phase for computing sa^* and lcp^* followed by an induction phase for inducing sa and lcp from sa^* and lcp^*. Suppose sa^* and lcp^* are already known, we can directly build the suffix and LCP arrays by calling the inducing process of an existing IS-based construction algorithm. This enlightens us to check the following conditions for verification:

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

(1) sa^* and lcp^* are both correct.
(2) $sa = sa'$ and $lcp = lcp'$, where sa' and lcp' are induced from sa^* and lcp^* by calling the inducing process of an existing IS-based construction algorithm.

Following the same idea described in Section 2.2, we come to the conclusion in Corollary 2 by using the fingerprinting technique.

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

(1) $sa^*[j] \neq sa^*[k]$.
(2) $\mathsf{fp}(sa^*[j], sa^*[j] + lcp^*[j] - 1) = \mathsf{fp}(sa^*[j-1], sa^*[j-1] + lcp^*[j] - 1)$.
(3) $x[sa^*[j] + lcp^*[j]] > x[sa^*[j-1] + lcp^*[j]]$.
(4) $sa[i] = sa'[i]$ and $lcp[i] = lcp'[i]$, where sa' and lcp' are induced from sa^* and lcp^* by calling the inducing process of an existing IS-based construction algorithm.

3.3 Algorithm

Algorithm 2 checks the conditions in Corollary 2 using external memory, the details are shown as following. The first task of the algorithm is to retrieve sa^* and lcp^* from sa and lcp. For the purpose, it creates a tuple for each suffix in sa and sorts them by 1st component in descending order (lines 2-3). After sorting, it scans x leftward to find all the S*-type suffixes according to the definition in Section 3.1 (lines 4-13). For each S*-type suffix, we pick the corresponding tuple from the top of ST_1 and forward the tuple to ST_2. Then, the algorithm sorts ST_2 by 2nd component in ascending order and scans the sorted tuples sequentially to produce sa^* and $rank^*$, where $rank*$ is the compact form of sa^*. Meanwhile, we compute lcp^* following the fact that the LCP-value of two suffixes in $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ $(i < j)$ is the minimum value among $\{lcp[i+1], ..., lcp[j-1], lcp[j]\}$ (lines 14-27). The next task is to check the correctness of sa^* and lcp^*. This is accomplished in lines 28-30 by reusing Algorithm 1. At last, we call the inducing process of an external-memory construction algorithm to generate sa' and lcp' (line 31), and compare the output with sa and lcp to determine the result in lines 32-36.

3.4 Optimization

Remember that, Algorithm 2 produces a copy of the suffix and LCP arrays during the inducing process. Actually, given that Σ is of a constant size, we can simply scan sa/lcp to induce and check suffixes/LCP arrays simultaneously without using extra space. The idea is that when a suffix/LCP-value is induced into a suffix/LCP bucket, we directly compare it with the corresponding value in sa/lcp. If the latter is equal to the induced one, then we believe it is correct. The key point here is how to retrieve elements from sa/lcp quickly. This can be accomplished by sequential I/O operations if we maintain a file pointer together with a reading buffer for each suffix/LCP sub-bucket. For better understanding, we show more details of the optimized inducing and checking processes below, where $c \in [0, \Sigma)$.

S1 Let $lp1[c]$ and $lp2[c]$ point to the leftmost element of $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{lcp\_bkt_L}(c)$, respectively. Induce L-type suffixes and their LCP-values. For each induced L-type suffix p with a heading character c_0 and its LCP-value q: (1) check $p = lp1[c_0]$ and $q = lp2[c_0]$; (2) let $lp1[c_0]$ and $lp2[c_0]$ point to the next element on the right.

S2 Let $sp1[c]$ and $sp2[c]$ point to the rightmost element of $\mathsf{sa\_bkt_S}(c)$ and $\mathsf{lcp\_bkt_S}(c)$, respectively. Induce S-type suffixes and their LCP-values. For each induced S-type

Algorithm 2: The Algorithm for checking the conditions of Corollary 2.

```
Function CheckByIS($x$, $sa$, $lcp$, $n$)
    $ST_1 := [(sa[i], i, null) | i \in [0, n)]$
    sort tuples in $ST_1$ by 1st component
    $r := 0, pos := -1$
    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
            $e.3rd := r$, $r := r + 1$, $ST_2$.push($e$), $pos := e.1st$
        end
    end
    sort tuples in $ST_2$ by 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
                $sa^*[j] := e.1st$, $rank^*[j] := e.3rd$, $lcp^*[j] := lcp_{min}$, $j := j + 1$, $i := i + 1$
                break
            end
            $i := i + 1$
        end
        $lcp_{min} := max\_val$
    end
    if CheckByFP$(x, rank^*, lcp^*, lcp^*$.size()$) = false$ 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
```

suffix p with a heading character c_0 and its LCP-value q: (1) check $p = sp1[c_0]$ and $q = sp2[c_0]$; (2) let $sp1[c_0]$ and $sp2[c_0]$ point to the next element on the left.

3.5 Analysis

Both Algorithm 2 can be implemented within sorting complexity. As will be shown in the next section, the space bottleneck of the algorithm occurs when inducing and checking the suffix and LCP arrays in lines 28-36 due to the space demand for storing BWT. Besides, our implementation of the optimized version has a similar I/O performance to that of Algorithm 1, while the former takes only half space in comparison with the latter.

4 Experiments

We implement the prototypes of Algorithms 1 and **??** in external memory. Our programs for the algorithms are compiled by gcc/g++ 4.8.4 with -O3 options under ubuntu 14.04 64-bit operating system running on a desktop computer, which is equipped with an Intel Xeon E3-1220 V2 CPU, 4GiB RAM and 500GiB HD. The following metrics normalized by the input's size are investigated for performance evaluation on the real-world datasets listed in Table 1:

- Running time: the running time in microseconds.
- Peak disk use: the peak use of external memory in bytes.
- I/O volume: the number of bytes read from and written to external memory.

4.1 Results

Some implementation choices in our programs are as follows. First, we use the second approach in Section **??** to facilitate the computation of fingerprints in Algorithm 1. Second, we employ the external-memory containers (vector, sorter and priority queue) provided by the STXXL library [28] to perform scans and sorts on disks. Finally, we use a 32-bit integer to represent each fingerprint and a 40-bit integer to represent each element in the suffix/LCP arrays.

TABLE 1
Corpus, n in Gi, 1 byte per character

Corpora	n	$\|\Sigma\|$	Description
enwiki	8	256	The 8-GiB prefix of an XML dump of English Wikipedia, available at https://dumps.wikimedia.org/enwiki, dated as 16/05/01.
uniprot	2.5	96	UniProt Knowledgebase, available at ftp://ftp.expasy.org/databases/.../complete, dated as 16/05/11.
proteins	1.1	27	Swissprot database, available at http://pizzachili.dcc.uchile.cl/texts/protein, dated as 06/12/15.

For description convenience, we denote the programs for Algorithms 1 and **??** by ProgA and ProgB, respectively. As shown in Figure 1, ProgA runs faster than ProgB on "enwiki", "uniprot" and "proteins" by about 20 percent. The speed gap between them is mainly due to the difference in their I/O efficiencies. Specifically, the I/O volume of ProgB is $190n$ in average, while that of ProgA is kept at $155n$ for different corpora. Notice that, although Algorithm **??** reuses Algorithm 1 to check sa_{LMS} and lcp_{LMS}, the consumption for the verification of sa_{LMS} and lcp_{LMS} in ProgB is at most half of ProgA because the number of LMS suffixes is no more than $\frac{1}{2}n$. It can be also observed that both programs are insensitive to the input corpus in terms of the space requirement. In details, the peak disk uses of ProgA and ProgB are respectively $26n$ and $40n$ on the three corpora.

We also investigate the performance trend of the two programs on the prefix of "enwiki" with the length varying on $\{1, 2, 4, 8\}$ GiB. Figure 2 illustrates that, as the prefix length increases, a performance degradation occurs to ProgB in both time and I/O efficiencies, but the fluctuation of ProgA can be ignored.

4.2 Discussions

It is identified that both programs heavily rely on the performance of the external memory sorter in use. A potential candidate for improving their speed is to adapt a GPU-based multi-way sorter (e.g., [29], [30]) for sorting massive data using external memory. By the aid of these fast sorting algorithms, the throughputs of the programs are expected to nearly approach the I/O bandwidth. Besides, the first two steps of Algorithm 1 are independent of each other and thus can be executed in parallel for acceleration. This technique can be also applied to check the suffix and LCP arrays of the LMS suffixes in Algorithm **??**.

Currently, for Algorithm **??**, step 2 constitutes the space bottleneck. It is worthy of mentioning that this step produces a copy of the suffix and LCP array during the inducing and checking processes. Actually, given that Σ is of a constant size and sa/lcp are known already, we can simply scan the input sa/lcp to perform the inducing process and compare each induced suffix/LCP value with that in the given sa/lcp to perform the checking process, resulting in less space consumption. To the end, we must maintain a read pointer for each suffix/LCP bucket in sa/lcp to scan elements in sequence.

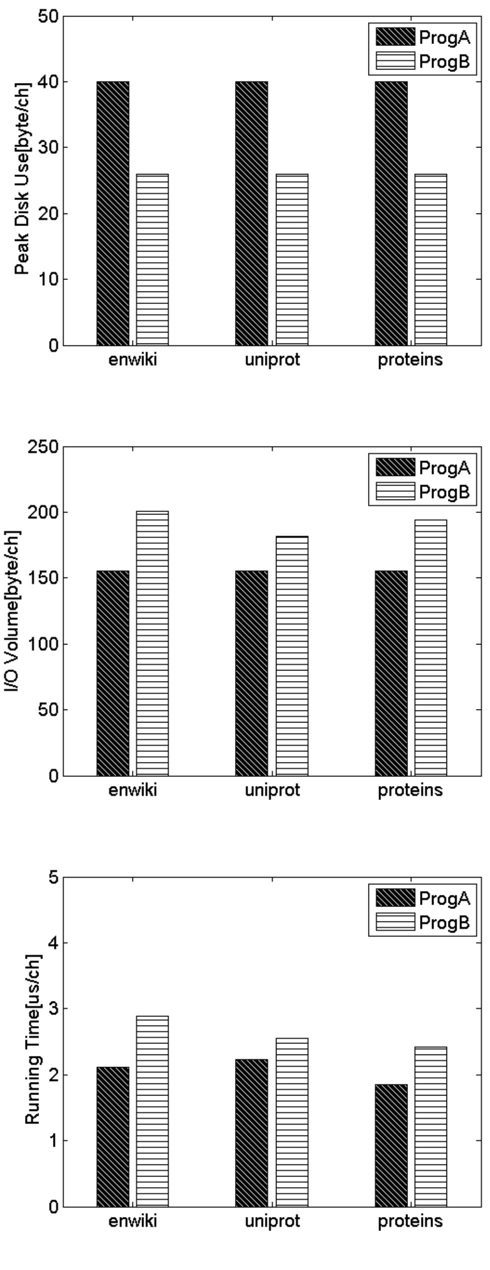

Fig. 1. Experimental results for various corpora.

5 Conclusions

We present two probabilistic methods for checking the suffix and LCP arrays in external memory using the Karp-

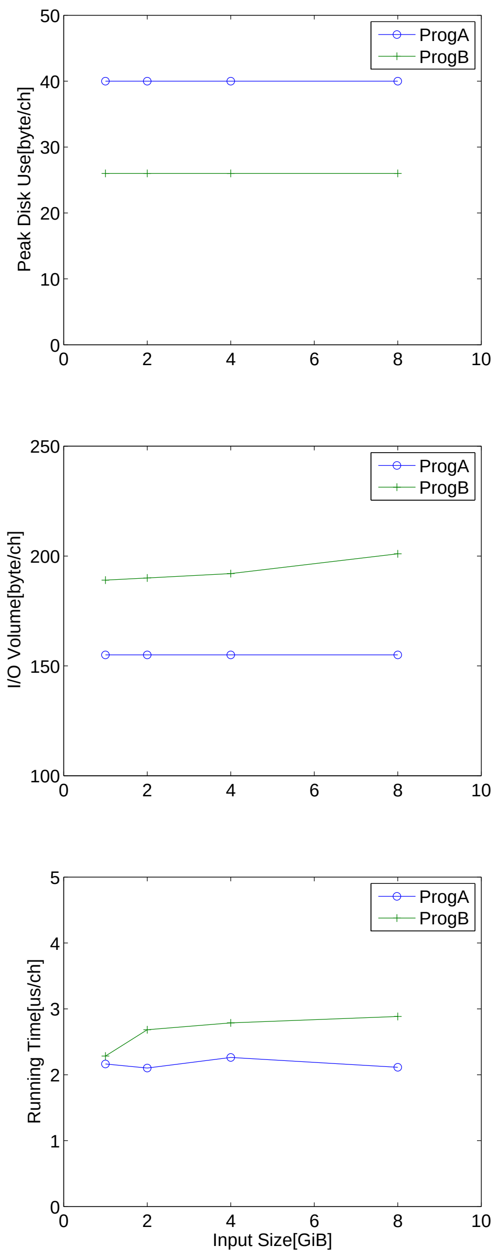

Fig. 2. Experimental results for prefixes of ”enwiki”.

Rabin fingerprinting function. Both methods can be employed to verify the suffix and LCP arrays after their construction, while the second method can be also integrated into an induced-sorting LACA to perform construction and verification in the same time.

We also design the algorithms for these two methods and implement the programs for performance evaluation. An experimental study is conducted to evaluate the time, space and I/O efficiencies. The results indicate that the program for the first method outperforms that for the second method in the running time and I/O volume by about 20 percent, while the peak disk use of the latter is about 26/40 = 0.65 as that of the former. There are still quite some opportunities to improve our algorithms and programs for better performance, for this our work has been undergoing.

References

[1] M. Abouelhodaa, S. Kurtzb, and E. Ohlebuscha, “Replacing Suffix Trees with Enhanced Suffix Arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, November 2004.

[2] U. Manber and G. Myers, “Suffix Arrays: A New Method for On-line String Searches,” *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[3] J. Kärkkäinen and P. Sanders, “Simple Linear Work Suffix Array Construction,” in *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, Eindhoven, Netherlands, June 2003, pp. 943–955.

[4] P. Ko and S. Aluru, “Space Efficient Linear Time Construction of Suffix Arrays,” in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Mexico, May 2003, pp. 200–210.

[5] D. K. Kim, J. S. Sim, H. Park, and K. Park, “Linear Time Construction of Suffix Arrays,” in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, June 2003, pp. 186–199.

[6] G. Nong, S. Zhang, and W. H. Chan, “Two Efficient Algorithms for Linear Time Suffix Array Construction,” *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, October 2011.

[7] R. Dementiev, J. Kärkäinen, J. Mehnert, and P. Sanders, “Better External Memory Suffix Array Construction.” *ACM Journal of Experimental Algorithmics*, vol. 12, no. 3, pp. 4:1–4:24, August 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 *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments*, 2012, pp. 88–102.

[11] J. Kärkkäinen and D. Kempa, “Engineering a Lightweight External Memory Suffix Array Construction Algorithm,” in *Proceedings of the 2nd International Conference on Algorithms for Big Data*, Palermo, Italy, April 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 Transactions on Information Systems*, vol. 32, no. 1, pp. 1:1–1:15, January 2014.

[13] G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu, “Induced Sorting Suffixes in External Memory,” *ACM Transactions on Information Systems*, vol. 33, no. 3, pp. 12:1–12:15, March 2015.

[14] V. Osipov, “Parallel Suffix Array Construction for Shared Memory architectures.” in *International Symposium on String Processing and Information Retrieval*, Cartagena de Indias, Colombia, October 2012, pp. 379–384.

[15] M. Deo and S. Keely, “Parallel Suffix Array and Least Common Prefix for the GPU,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York ,USA, August 2013, pp. 197–206.

[16] L. Wang, S. Baxter, and J. Owens, “Fast Parallel Suffix Array on the GPU,” in *In proceedings of the 21st International Conference on Parallel and Distributed Computing*, August 2015, pp. 573–587.

[17] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, “Parallel External Memory Suffix Sorting,” in *In proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching*, Ischia Island, Italy, July 2015, pp. 329–342.

[18] J. Fischer, “Inducing the LCP-Array,” in *Algorithms and Data Structures*, ser. Lecture Notes in Computer Science, 2011, vol. 6844, pp. 374–385.

[19] P. Flick and S. Aluru, “Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays,” in *In proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, USA, 2015, pp. 1–10.

[20] T. K. G. Lee, H. Arimura, S. Arikawa, and K. Park, “Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its Applications,” in *In proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Jerusalem, Israel, July 2001, pp. 181–192.

[21] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, “Permuted Longest-Common-Prefix Array,” in *Proceedings of the 20th Annual Symposuim on Combinatorial Pattern Matching*, Lille, France, June 2009, pp. 181–192.

[22] S. J. Puglisi and T. Andrew, “Space-time Tradeoffs for Longest-Common-Prefix Array Computation,” in *In proceedings of the 19th International Symposium on Algorithms and Computation*, Gold Coast, Australia 2008, pp. 124–135.

[23] J. Kärkkäinen and D. Kempa, “Faster External Memory LCP Array Construction,” in *International Proceedings in Informatics*, 2016.

[24] P. Bille, J. Fischer, and et al., “Sparse Suffix Tree construction in Small Space,” in *In proceedings of the International Colloquium on Automata, Languages, and Programming*, 2013, pp. 148–159.

[25] S. Burkhardt and J. Kärkkäinen, “Fast Lightweight Suffix Array Construction and Checking,” in *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, Morelia, Mexico, May 2003, pp. 55–69.

[26] R. Karp and M. Rabin, “Efficient Randomized Pattern Matching Algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, March 1987.

[27] L. Arge and M. Thorup, “RAM-efficient external memory sorting.” *Algorithms and Computations*, vol. 9293, no. 3, pp. 491–501, 2013.

[28] R. Dementiev, L. Kettner, and P. Sanders, “STXXL: Standard Template Library for XXL Data Sets,” *Software: Practice and Experience*, vol. 38, no. 6, pp. 589–637, 2008.

[29] N. Leischner, V. Osipov, and P. Sanders, “GPU Sample Sort,” in *In proceedings of the International Symposium on Parallel and Distributed Processing*, Atlanta, USA, 2010, pp. 1–10.

[30] A. Davidson, D. Garland, and et al., “Efficient Parallel Merge Sort for Fixed and Variable Length Keys,” in *In proceedings of the International Symposium on Innovative Parallel Computing*, California, USA, 2012, pp. 1–9.