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 [**?**], 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 [**?**]. From then on, a plethora of SACAs have been proposed on different computation models, e.g., internal memory [**?**], [**?**], [**?**], [**?**], external memory [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] and shared memory models [**?**], [**?**], [**?**], [**?**]. 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 [**?**], [**?**], [**?**] and that of the second category take the suffix array (SA) and/or Burrows-Wheeler transform (BWT) as input to facilitate the computation [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].

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 [**?**]. 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 [**?**] 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 mainly 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.

By using the same fingerprinting technique, the second method first verifies a subset chosen from the input arrays

- *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.

and then produces a copy of the suffix and LCP arrays from the verified subset following the induced sorting 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 drawn from an alphabet Σ, we assume that $x[n-1] = \$$ and $\$ \in \Sigma$ is smaller than any other characters in x. The suffix array of x, denoted by sa, is a permutation of $\{0, 1, ..., n-1\}$ such that $\mathsf{suf}(sa[i]) < \mathsf{suf}(sa[j])$ for all $0 \leq i < j < n$, where $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ are the suffixes starting with $x[sa[i]]$ and $x[sa[j]]$, respectively. Particularly, we say $\mathsf{suf}(sa[i])$ is a lexical neighbor of $\mathsf{suf}(sa[j])$ if $|i-j| = 1$. The LCP array of x, denoted by lcp, consists of n integers, where $lcp[i]$ records the LCP-value of $\mathsf{suf}(sa[i])$ and its left neighbor in sa for $i \in [1, n)$.

2.2 Idea

The lexical order and the LCP-value of two suffixes starting with $x[i]$ and $x[j]$ can be determined by literally comparing their characters rightward with k increasing from 0. 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 $x[i+k] \neq x[j+k]$. Lemma 1 indicates that 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 takes at most $\mathcal{O}(n)$ character-wise comparisons. 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. Then, the equality of two substrings can be tested in constant time by comparing the corresponding hash values instead.

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. □

The key point here is how to efficiently compute the hash values for the involved substrings. Taking into account the high cost of finding a PHF for this requirement, we choose to use the Karp-Rabin fingerprinting function **[?]** for conducting these conversions, where all the substrings can be transformed into their integer forms, called fingerprints, by accessing characters of x sequentially. Specifically, suppose L is a prime and δ is randomly chosen from $[1, L)$, the fingerprint $\mathsf{fp}(i, j)$ of substring $x[i, j]$ can be calculated according to the three formulas below, where the first two are for computing $\mathsf{fp}(0, i-1)$ and $\mathsf{fp}(0, j)$ when scanning x rightward and the last is for computing their difference. It is worthy of pointing out that two equal substrings always have a common fingerprint, but the inverse is not true. Fortunately, it has been proved that the probability of a false match can be reduced to a negligible level by setting L to a large value **[?]**. This property is utilized in **[?]** to design a probabilistic algorithm for computing a sparse suffix array.

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

The above discussion leads us to Corollary 1. In the rest of this section, we will describe the algorithmic design and implementation of our first checking method based on this 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

We design Algorithm 1 to check the conditions in Corollary 1. This algorithm mainly consists of three steps and each step is explained detailedly in the following paragraphs.

Step 1. The first step can be divided into three substeps:

(1a) Scan sa and lcp rightward with i increasing from 1 to $n-1$. For each scanned $\langle sa[i], lcp[i] \rangle$, create an AITEM with $idx = i, pos = sa[i], lv = lcp[i]$. Sort the items by pos in ascending order.

(1b) Scan x rightward with p increasing from 0 to $n-1$. For each scanned $x[p]$, compute $\mathsf{fp}(0, p)$ and update all the AITEMs with $pos = p+1$ by setting $fp = \mathsf{fp}(0, p)$ and $pos = pos + lv$. Sort the items by pos in ascending order.

(1c) Scan x rightward with p increasing from 0 to $n-1$. For each scanned $x[p]$, compute $\mathsf{fp}(0, p)$ and update all the AITEMs with $pos = p+1$ by setting $fp = \mathsf{fp}(0, p) - fp \cdot \delta^{lv} \, mod \, L$ and $ch = x[p+1]$. Sort the items by idx in ascending order.

An auxiliary data structure called AITEM is introduced to facilitate the computation of fingerprints, where each AITEM is a 5-tuple $\langle idx, pos, lv, ch, fp \rangle$ for a suffix. To enable fast calculation of $\mathsf{fp}(sa[i]-1)/\mathsf{fp}(sa[i]+lcp[i]-1)$ for all $i \in [1, n)$, step 2 first sorts the AITEMs by pos and then sequentially retrieves $\mathsf{fp}(0, pos)$ for each item by iteratively computing $\{\mathsf{fp}(0, 0), \mathsf{fp}(0, 1), \ldots, \mathsf{fp}(0, n-1)\}$ according to Formula 1. Given that $\mathsf{fp}(sa[i]-1)$ and $\mathsf{fp}(sa[i]+lcp[i]-1)$ are already known, $\mathsf{fp}(sa[i], sa[i]+lcp[i]-1)$ is obtained according to Formula 2.

Notice that all the AITEMs are sorted by the starting positions of the corresponding suffixes in substep (1a). Thus,

Algorithm 1: The algorithm using Method A.

Input: x, sa, lcp
Step 1. Compute $\mathsf{fp}(sa[i], sa[i] + lcp[i] - 1)$ and retrieve $x[sa[i] + lcp[i]]$ for $i \in [1, n)$.
Step 2. Compute $\mathsf{fp}(sa[i], sa[i] + lcp[i+1] - 1)$ and retrieve $x[sa[i] + lcp[i+1]]$ for $i \in [0, n-1)$.
Step 3. Check if $\mathsf{fp}(sa[i], sa[i] + lcp[i] - 1) = \mathsf{fp}(sa[i-1], sa[i-1] + lcp[i] - 1)$ and $x[sa[i] + lcp[i]] > x[sa[i-1] + lcp[i]]$ for $i \in [1, n)$.

we check condition (1) of Corollary 1 during the time when sequentially accessing the items in substep (1b).

Step 2. This step computes $\mathsf{fp}(sa[i], sa[i]+lcp[i+1]-1)$ for $i \in [0, n-1)$ using the following three substeps:

(2a) Scan sa and lcp rightward with i increasing from 0 to $n-2$. For each scanned $\langle sa[i], lcp[i+1]\rangle$, create an AITEM with $idx = i, pos = sa[i], lv = lcp[i+1]$. Sort the items by pos in ascending order.

(2b) Reuse step (1b) to update and sort AITEMs.

(2c) Reuse step (1c) to update and sort AITEMs.

The major difference between the first two steps lies in that, the field lv of the AITEMs for $\mathsf{suf}(sa[i])$ are respectively set as $lcp[i]$ and $lcp[i+1]$ in substeps (1a) and (2a).

Step 3. Let $arrA$ and $arrB$ denote the arrays of AITEMs produced by steps 1 and 2, respectively. Scan $arrA$ and $arrB$ rightward with i increasing from 0 to $n-2$. For each pair $\langle arrA[i], arrB[i]\rangle$, check if $arrA[i].fp = arrB[i].fp$ and $arrA[i].ch > arrB[i].ch$ to ensure the correctness of conditions (2)-(3) in Corollary 1.

2.4 Substring Fingerprint

Algorithm 1 computes $\mathsf{fp}(i, j)$ from $\mathsf{fp}(0, i-1)$ and $\mathsf{fp}(0, j)$ according to Formula 2. Let e denote $j - i + 1$, then the coefficient $(\delta^e \mod L)$ must be calculated quickly or it will become a performance bottleneck. One approach first decomposes e into $k_1 \cdot m + k_2$ and then computes δ^e by multiplying $\delta^{k_1 \cdot m}$ and δ^{k_2}, where k_1, k_2 are integers and $k_1 \in [0, \lceil \frac{n}{m} \rceil], k_2 \in [0, m)$. An alternative 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 $k_i \in \{0, 1\}$. The first approach answers in $O(1)$ time and consumes $\mathcal{O}(m + \lceil \frac{n}{m} \rceil)$ space to maintain two lookup tables for storing $\{\delta^0, \delta^m, \ldots, \delta^{\lceil \frac{n}{m} \rceil m}\}$ and $\{\delta^0, \delta^1, \ldots, \delta^{m-1}\}$, respectively, while the second approach answers in $\mathcal{O}(\lceil \log 2^n \rceil)$ time and consumes $\mathcal{O}(\lceil \log 2^n \rceil)$ space to store $\{\delta^1, \delta^2, \ldots, \delta^{2^{\lceil \log 2^n \rceil}}\}$. For a better space performance, we adopt the second approach to compute the fingerprints in our implementation for the algorithm.

2.5 Analysis

Algorithm 1 performs multiple scans and sorts for fixed-size objects with integer keys. 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 an integer sort are $\mathcal{O}(n \log_{M/B}(n/B))$ and $\mathcal{O}((n/B) \log_{M/B}(n/B))$, respectively **[?]**. Besides, the algorithm reaches its peak disk use when sorting n AITEMs in external memory during the second step. Given that each input character occupies a words and each fingerprint occupies b words, the space consumption for each sort is $(a + b + 3)n$ words.

2.6 Example

3 Method B

As we will see from the experimental study in Section 4, Algorithm 1 designed by Method A is quite space consuming, the peak disk use is 40 bytes per input character. This motivates us to design a lightweight checker that uses less space. In this section, we propose a method for checking the suffix and LCP arrays using the induced-sorting principle. The experimental results show that Algorithm 3 designed by this method consumes about 40% less space than Algorithm 1 on real-world data.

3.1 Notations

Assume $x[n-1]$ is lexicographically the smallest character in Σ and appears in $x[0, n)$ only once, we introduce the following notations and symbols for description clarity.

L-type/S-type/LMS character and suffix. All the characters in x are classified into three different types, namely L-type, S-type and LMS. In details, $x[i]$ is S-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 S-type; otherwise, $x[i]$ is L-type. In particular, if $x[i-1]$ and $x[i]$ are respectively L-type and S-type, then $x[i]$ is LMS. Furthermore, $\mathsf{suf}(i)$ is L-type, S-type or LMS if $x[i]$ is L-type, S-type or LMS, respectively.

L-type/S-type/LMS substring. Suppose $x[j]$ is LMS and no characters in $x[i+1, j-1]$ are LMS, then substring $x[i, j]$ is an L-type, S-type or LMS substring if $x[i]$ is L-type, S-type or LMS, respectively.

Suffix buckets. Suffixes in sa are naturally grouped into multiple buckets, where each bucket occupies a contiguous interval of sa and contains all the suffixes with an identical heading character. Besides, a bucket can be further divided into two parts, where the left/right part is a subbucket containing only L-type/S-type suffixes. For short, we use $\mathsf{sa\_bkt}(c)$ to denote the bucket containing all the suffixes with a heading character c, and $\mathsf{sa\_bkt_L}(c)$/$\mathsf{sa\_bkt_S}(c)$ to denote the left/right subbucket in $\mathsf{sa\_bkt}(c)$.

LCP buckets. Suppose $\mathsf{sa\_bkt}(c)$ is a suffix bucket ranging from $sa[i]$ to $sa[j]$, then $lcp[i, j]$ is the corresponding LCP bucket, denoted by $\mathsf{lcp\_bkt}(c)$, and it can be further divided into two subbuckets $\mathsf{lcp\_bkt_L}(c)$ and $\mathsf{lcp\_bkt_S}(c)$ according to $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{sa\_bkt_S}(c)$.

Character repetition count. Suppose $x[i] = x[i+1] = \cdots = x[j]$ and $x[j] \neq x[j+1]$, then $j - i + 1$ is the number of successive repetitions for $x[i]$. Let $\mathsf{rep}(i)$ denote the value.

Inverse suffix array. The inverse suffix array isa satisfies that $isa[sa[i]] = i$ for $i \in [0, n)$.

Reduced string. The reduced string x_1 records the names of LMS suffixes, where each name represents the relative order of the corresponding suffix among all the LMS ones.

Position array. The position array pa records the starting positions of LMS suffixes.

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

In addition, we denote the suffix and LCP arrays of LMS suffixes by sa_{LMS} and lcp_{LMS}.

3.2 An Overview of Inducing LCP Array

An induced-sorting SACA (e.g., **[?]**) mainly consists of a reduction phase followed by an induction phase. In brief, it first sorts and names the LMS substrings to produce the reduced string x_1 during the reduction phase and then determines whether there exist two equal characters in x_1. If yes, then it recursively performs the reduction phase by replacing x with x_1; otherwise, it computes sa_1 from x_1 and performs the induction phase to induce sa from sa_1. The work presented in **[?]** shows that an induced-sorting SACA can be modified to design an LACA for constructing the suffix and LCP arrays simultaneously, which principle is explained here for readers to understand our method. Its main idea is to compute the LCP value of two suffixes placed at the neighboring positions in the same subbucket according to the following properties, where $\mathsf{suf}(sa[j])$ and $\mathsf{suf}(sa[k])$ are the suffixes from which induce $\mathsf{suf}(sa[i-1])$ and $\mathsf{suf}(sa[i])$ during the induction phase.

Property 1. If $x[sa[i]] = x[sa[i-1]]$, $t[sa[i]] = t[sa[i-1]]$ and $x[sa[j]] \neq x[sa[k]]$, then $lcp[i] = 1$.

Property 2. If $x[sa[i]] = x[sa[i-1]]$, $t[sa[i]] = t[sa[i-1]]$ and $x[sa[j]] = x[sa[k]]$, then $lcp[i] = 1 + \mathsf{min}(lcp[j+1], lcp[j+2], \ldots, lcp[k])$.

Algorithm 2 summarizes the main steps of the induction phase of the induced-sorting LACA in **[?]**. Suppose sa_{LMS} and lcp_{LMS} have been computed and their elements are inserted into the corresponding buckets in sa and lcp, the last two steps of the algorithm are described below.

S3. Clear $\mathsf{sa\_bkt_L}(c)/\mathsf{lcp\_bkt_L}(c)$ for $c \in \Sigma$ in sa/lcp.
  (1) Scan sa rightward with i increasing from 0 to $n-1$. For each scanned item $sa[i]$, insert $sa[i]-1$ into the leftmost empty position of $\mathsf{sa\_bkt_L}(x[sa[i]-1])$ if $\mathsf{t}[sa[i]-1] = 0$.
  (2) For each induced item placed in $sa[i]$: set $lcp[i] = 0$ if $x[sa[i]] \neq x[sa[i-1]]$; otherwise, compute $lcp[i]$ following Properties 1-2.

S4. Clear $\mathsf{sa\_bkt_S}(c)/\mathsf{lcp\_bkt_S}(c)$ for $c \in \Sigma$ in sa/lcp.
  (1) Scan sa leftward with i decreasing from $n-1$ to 0. For each scanned item $sa[i]$, insert $sa[i]-1$ into the rightmost empty position of $\mathsf{sa\_bkt_S}(x[sa[i]-1])$ if $\mathsf{t}[sa[i]-1] = 1$.
  (2) For each induced item placed in $sa[i-1]$: if $x[sa[i]] = x[sa[i-1]]$ and $t[sa[i]] = t[sa[i-1]]$, then compute $lcp[i]$ following Properties 1-2.

There remain two special cases needing to be handled separately. When inducing L-type suffixes in S3, the LCP value of the rightmost L-type and the leftmost LMS suffixes in the same bucket, say $\mathsf{suf}(sa[i_1])$ and $\mathsf{suf}(sa[i_2])$, is equal to $\mathsf{min}(\mathsf{rep}(sa[i_1]), \mathsf{rep}(sa[i_2]))$. Likewise, when inducing S-type suffixes in S4, the LCP value of the rightmost L-type and the leftmost S-type suffixes in the same bucket is computed following the same way. Notice that, $\mathsf{rep}(sa[i_1])/\mathsf{rep}(sa[i_2])$ is calculated by literally comparing the characters in the suffix from left to right. It is revealed in **[?]** that, a repetition interval cannot stride over two successive LMS substrings, thus all the involved comparisons can be done in overall linear running time.

3.3 Check LCP Array

The inducing process of the LCP array in Section 3.2 also constitutes the sufficient and neccessary conditions for a correct lcp as stated below.

Lemma 2. Given that sa is correct, lcp is correct if and only if:

(1) lcp_{LMS} is correct.
(2) $lcp[i]$ is equal to the value calculated by the inducing process in Section 3.2, for $i \in [0, n)$.

This lemma suggests a method to check an LCP array as follows. We address two cases: (1) given lcp_{LMS} and correct sa, to build and check lcp simultaneously; (2) given lcp and correct sa, to check lcp.

Notice that the processes for inducing and checking lcp are essentially the same in terms of that both utilize condition (2) in Lemma 2, the only difference is that the former computes $lcp[i]$ using the condition but the latter checks if the condition is held for each given $lcp[i]$.

Case 1. In the first case, we first check lcp_{LMS} by Method A, then induce and check lcp from lcp_{LMS} simultaneously. The problem to be solved is that when an item is induced into a bucket, its corresponding item $lcp[i]$ to be compared is not known yet. Our solution for this is to integrate the processes of inducing and checking into a single one, by ensuring the sequence of items induced into a bucket in lcp in the inducing process is identical to that later seen by the scanning process that is also the checking process. For each bucket, we increasingly compute the fingerprints of both sequences and check their equality during the whole process for inducing lcp. If the two fingerprints for each bucket are equal, then the condition (2) of Lemma 2 will be seen by the checking process with a high probability. As a result, lcp can be built and probabilistically checked in the same time using this method, for this we have Corollary 2.

Corollary 2. Given that sa is correct, then lcp is correct with a high probability if and only if these conditions are satisfied for all $c \in \Sigma$:
(1) lcp_{LMS} is correct.
(2) The fingerprints of $lcp_{\mathsf{L1}}(c)$ and $lcp_{\mathsf{L2}}(c)$ are equal.
(3) The fingerprints of $lcp_{\mathsf{S2}}(c)$ and $lcp_{\mathsf{S2}}(c)$ are equal.

For each c, $lcp_{\mathsf{L2}}(c)/lcp_{\mathsf{S2}}(c)$ or $lcp_{\mathsf{L1}}(c)/lcp_{\mathsf{S1}}(c)$ records the sequences of items in $\mathsf{lcp\_bkt_L}(c)/\mathsf{lcp\_bkt_S}(c)$ in their order of being scanned or induced, respectively. We augment S3 and S4 of Algorithm 2 to compute the fingerprints of these sequences as follows:

S3'. Initialize $\mathsf{fp}(lcp_{\mathsf{L1}}(c)) = 0$ and $\mathsf{fp}(lcp_{\mathsf{L2}}(c)) = 0$.
  (1) For each scanned item $sa[i]$, update $\mathsf{fp}(lcp_{\mathsf{L2}}(c))$ by $lcp[i]$ if $x[sa[i]] = c$ and $t[sa[i]] = 0$.
  (2) For each induced item placed in $sa[i]$, update $\mathsf{fp}(lcp_{\mathsf{L1}}(c))$ by lv if $x[sa[i]] = c$, where lv is the calculated LCP value.

Algorithm 2: The algorithm of the induction phase for an induced-sorting LACA.

Input: x, sa_1, lcp_1
Output: sa, lcp
S1. Compute sa_{LMS} and lcp_{LMS} from sa_1 and lcp_1, respectively.
S2. Insert items of sa_{LMS} and lcp_{LMS} into sa and lcp.
S3. Induce L-type suffixes and LCP lengths.
S4. Induce S-type suffixes and LCP lengths.

S4′. Initialize $\mathsf{fp}(lcp_{\mathsf{S1}}(c)) = 0$ and $\mathsf{fp}(lcp_{\mathsf{S2}}(c)) = 0$.
(1) For each scanned item $sa[i]$, update $\mathsf{fp}(lcp_{\mathsf{S2}}(c))$ by $lcp[i]$ if $x[sa[i]] = x[sa[i-1]] = c$ and $t[sa[i]] = t[sa[i-1]] = 1$.
(2) For each induced item placed at $sa[i-1]$, update $lcp_{\mathsf{S1}}(c)$ by lv if $x[sa[i-1]] = x[sa[i]] = c$ and $t[sa[i-1]] = t[sa[i]] = 1$, where lv is the calculated LCP value.

We generalize Formula 1 in Section 2.2 to iteratively compute the fingerprint of each sequence in steps S3′ and S4′. Specifically, when a new item e is appended to a sequence, its fingerprint fp is updated according to the formula $fp = fp \cdot \delta + e \mod L$.

Case 2. In the second case, we first scan lcp to compute lcp_{LMS} and check lcp_{LMS} by Method A, then induce and check the LCP array from lcp_{LMS} following the same idea of the first case. After that, we compare the given lcp with the induced LCP array to ensure the correctness of the former.

3.4 Check Suffix Array

An induced-sorting SACA determines the order of two suffixes by comparing their heading characters and succeeding suffixes in x. In other words, Property 3 is held for any two suffixes placed in $sa[i]$ and $sa[j]$, where $\mathsf{suf}(sa[p])$ and $\mathsf{suf}(sa[q])$ are the two suffixes from which induce $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ during the induction phase.

Property 3. If $(x[sa[i]], p) > (x[sa[j]], q)$, then $i > j$.

As shown below, Property 3 can be used to constitute the sufficient and necessary conditions for a correct sa.

Lemma 3. sa is correct if and only if:

(1) sa_{LMS} is correct.
(2) $i > j$ if $(x[sa[i]], p) > (x[sa[j]], q)$, for $i, j \in [0, n)$.

We use Lemma 3 to check a suffix array in two cases: (1) given sa_{LMS}, to build and check sa in the same time; (2) given sa, to check sa.

Case 1. In the first case, we first check sa_{LMS} by using Method A, and then induce and check sa from sa_{LMS} by comparing the fingerprint of the sequence of items induced into a bucket with that of the sequence of items scanned in the bucket. If the two fingerprints are equal for each bucket, then the condition (2) of Lemma 3 are satisfied with a high probability. Following the discussion, we have Corollary 3.

Corollary 3. sa is correct with a high probability if and only if these conditions are satisfied for all $c \in \Sigma$:
(1) sa_{LMS} is correct.
(2) The fingerprints of $sa_{\mathsf{L1}}(c)$ and $sa_{\mathsf{L2}}(c)$ are equal.
(3) The fingerprints of $sa_{\mathsf{S2}}(c)$ and $sa_{\mathsf{S2}}(c)$ are equal.

For each c, $sa_{\mathsf{L2}}(c)/sa_{\mathsf{S2}}(c)$ or $sa_{\mathsf{L1}}(c)/sa_{S1}(c)$ records the sequence of items in $\mathsf{sa\_bkt_L}/\mathsf{sa\_bkt_S}$ in their order of being scanned or induced, respectively. Similar to S3′ and S4′, we compute the fingerprints of these sequences by augmenting the last two steps of Algorithm 2 as follows:

S3″. Initialize $\mathsf{fp}(sa_{\mathsf{L1}}(c)) = 0$ and $\mathsf{fp}(sa_{\mathsf{L2}}(c)) = 0$.
(1) For each scanned item $sa[i]$, update $\mathsf{fp}(sa_{\mathsf{L2}}(c))$ by $sa[i]$ if $x[sa[i]] = c$ and $t[sa[i]] = 0$.
(2) For each induced item, update $\mathsf{fp}(sa_{\mathsf{L1}}(c))$ by $sa[j]-1$ if $x[sa[j]-1] = c$, where $sa[j]$ is the starting position of the suffix from which induces the item.

S4″. Initialize $\mathsf{fp}(sa_{\mathsf{S1}}(c)) = 0$ and $\mathsf{fp}(sa_{\mathsf{S2}}(c)) = 0$.
(1) For each scanned item $sa[i]$, update $\mathsf{fp}(sa_{\mathsf{S2}}(c))$ by $sa[i]$ if $x[sa[i]] = c$ and $t[sa[i]] = 1$.
(2) For each induced item, update $\mathsf{fp}(sa_{\mathsf{S1}}(c))$ by $sa[j]-1$ if $x[sa[j]-1] = c$, where $sa[j]$ is the starting position of the suffix from which induces the item.

Case 2. In the second case, we first scan sa to compute sa_{LMS} and check sa_{LMS} by using Method A, then induce and check the suffix array from sa_{LMS} following the same idea of the first case. Afterward, we compare the given sa with the induced suffix array to ensure the correctness of the former.

3.5 Algorithms

We propose two external-memory algorithms based on the method described in Sections 3.3-3.4. The first algorithm can build and check the suffix and LCP arrays simultaneously, while the second algorithm, called Algorithm 3, can check the two arrays after their construction.

3.5.1 Build and Check Simultaneously

We adapt Algorithm 2 to check the conditions of Corollaries 2-3 when building the suffix and LCP arrays by an induced-sorting LACA.

There are three places needed to be revised in Algorithm 2. The first place is to check sa_{LMS} and lcp_{LMS} using Algorithm 1 after computing them in S1. The second place is to compute the fingerprints by adapting S3 and S4 of Algorithm 2 as below:

(1) Augment S3 with S3′ and S3″ to compute the fingerprints of $\mathsf{fp}(lcp_{\mathsf{L1}}(c)/lcp_{\mathsf{L2}}(c))$ and $\mathsf{fp}(sa_{\mathsf{L1}}(c)/sa_{\mathsf{L2}}(c))$ for all $c \in \Sigma$.
(2) Augment S4 with S4′ and S4″ to compute the fingerprints of $\mathsf{fp}(lcp_{\mathsf{S1}}(c)/lcp_{\mathsf{S2}}(c))$ and $\mathsf{fp}(sa_{\mathsf{S1}}(c)/sa_{\mathsf{S2}}(c))$ for all $c \in \Sigma$.

The third place is to compare the fingerprints after the inducing process.

This algorithm can be applied to the external memory model if we extend Algorithm 2 for inducing the suffix and LCP arrays using external memory by the method described in [**?**].

3.5.2 Build and Check Sequentially

Algorithm 3 mainly consists of three steps. In the first step, it computes sa_{LMS} and lcp_{LMS} from sa and lcp, and ensures their correctness by using Algorithm 1. In the second step, it checks the conditions of Corollaries 2-3 to verify the suffix and LCP arrays induced from sa_{LMS} and lcp_{LMS}. In the last step, it checks the given sa and lcp by comparing them with the induced suffix and LCP arrays. In what follows, we explain the algorithm step by step in more details for better understanding.

Algorithm 3: The algorithm that performs verification after construction using Method B.

Input: x, sa, lcp
Step 1. Compute and check sa_{LMS} and lcp_{LMS}.
Step 2. Induce and check the suffix and LCP arrays from sa_{LMS} and lcp_{LMS}.
Step 3. Check sa and lcp by comparing them with the induced suffix and LCP arrays.

Step 1. This step computes sa_{LMS} and lcp_{LMS} from sa and lcp by the following three substeps:

(1a) Sort i by $sa[i]$ for $i \in [0, n)$ to produce isa.
(1b) Sort $\langle x[i-1], t[i-1], x[i], t[i], \mathsf{rep}(i)\rangle$ by $isa[i]$ for $i \in [0, n)$ to produce an array arr, where $arr[i]$ records $\langle x[sa[i]-1], t[sa[i]-1], x[sa[i]], t[sa[i]], \mathsf{rep}(sa[i])\rangle$.
(1c) Initialize $v = n$. Scan sa, lcp and arr rightward with i increasing from 0 to $n-1$. For each scanned $\langle sa[i], lcp[i], arr[i]\rangle$: (1) if $lcp[i] < v$, then set $v = lcp[i]$; (2) if $t[sa[i]-1] = 0$ and $t[sa[i]] = 1$, then append $sa[i]$ and v to sa_{LMS} and lcp_{LMS}, respectively, and reset $v = n$.

As can be seen, step 1 determines whether or not a suffix is LMS by checking the types of its preceding and heading characters. To avoid random accesses to x residing on disks, it first sorts the preceding and heading characters of suffixes by isa to produce arr, and then scans sa, lcp and arr to identify all the LMS suffixes. During the process, for any two successively found LMS suffixes, say $\mathsf{suf}(sa[i_1])$ and $\mathsf{suf}(sa[i_2])$, the LCP length of them is calculated by tracing the minimum of $lcp[i_1+1, i_2]$. After that, sa_{LMS} and lcp_{LMS} are checked by using Algorithm 1.

Notice that, when inducing in external memory, the heading characters of successively induced suffixes should be loaded into RAM by sequential I/O operations. For the purpose, the preceding characters calculated by substeps (1a)-(1b) are stored in external memory to facilitate the inducing process in step 2. Specifically, when scanning arr in substep (1c), we distribute the elements of the array into $arr_{\mathsf{L}}, arr_{\mathsf{S}}$ and arr_{LMS} as following: append $arr[i]$ to arr_{L} or arr_{S} if $t[sa[i]] = 0$ or 1, respectively, and append $arr[i]$ to arr_{LMS} if $t[sa[i]-1] = 0$ and $t[sa[i]] = 1$.

Step 2. This step employs two priority queues PQ_{L} and PQ_{S} to emulate the RAM-based inducing process in external memory, where each element of both priority queues is a 6-tuple $\langle ch, ty, ra, pos, lv, rp\rangle$ defined as below:

- pos: starting position of $\mathsf{suf}(pos)$.
- ch: heading character of $\mathsf{suf}(pos)$.
- ty: type of $\mathsf{suf}(pos)$.
- ra: rank of $\mathsf{suf}(pos)$.
- lv: LCP length of $\mathsf{suf}(pos)$ and its left or right neighbor in the induced SA.
- rp: $\mathsf{rep}(pos)$.

At the very beginning, step 2 first scans sa_{LMS} and lcp_{LMS} to put the sorted LMS suffixes along with the corresponding LCP values into PQ_{L} with the sorting key $\langle ch, ty, ra\rangle$, then it pops the elements of PQ_{L} in ascending order to induce L-type suffixes. Specifically, for each visited element e, it retrieves the preceding character of $\mathsf{suf}(e.pos)$ from the leftmost unvisited element in arr_{L} or arr_{LMS} if $e.ty = 0$ or 1, respectively. If the retrieved preceding character is L-type, then it computes the LCP length of $\mathsf{suf}(pos-1)$ and its left neighbor in the induced SA, and puts the induced suffix along with the calculated LCP value into PQ_{L}. Meanwhile, it puts $\mathsf{suf}(e.pos)$ into PQ_{S} if $e.ty = 0$.

When PQ_{L} is empty, all the L-type suffixes are sorted in PQ_{S} with the sorting key $\langle ch, ty, ra\rangle$. Step 2 continues to pop the elements of PQ_{S} in descending order to induce S-type suffixes and their LCP values. For each visited element e, it retrieves the preceding character of $\mathsf{suf}(e.pos)$ from the rightmost unvisited element in arr_{L} or arr_{S} if $e.ty = 0$ or 1, respectively. If the preceding character is S-type, then it calculates the LCP length of $\mathsf{suf}(e.pos-1)$ and its right neighbor in the induced SA, and puts the induced suffix along with the calculated LCP value into PQ_{S}. Meanwhile, it outputs $e.pos$ and $e.lv$ to build the suffix and LCP arrays.

When PQ_{S} is empty, the construction of the suffix and LCP arrays is completed and the calculation of the required fingerprints is also finished. Then, step 2 compares the fingerprints to check the correctness of the induced suffix and LCP arrays.

Notice that, we reuse a technique proposed in [**?**] to compute the LCP value of two neighboring suffixes in a sub-bucket according to Properties 1-2. Specifically, we record the heading characters and the LCP value of two suffixes from which most recently induce a suffix into $\mathsf{sa\_bkt}(c)$, for all $c \in \Sigma$. Given that Σ is of a constant size, it takes overall $\mathcal{O}(n)$ time to keep the data structure up-to-date. Besides, for the rightmost L-type and the leftmost S-type/LMS suffixes in a bucket, we compare the repetition counts of their heading characters to compute the LCP length of them.

Step 3. This step performs a scan on sa and lcp to compare the elements of them with those of the induced suffix and LCP arrays.

3.6 Analysis

Clearly, Algorithm 3 can be implemented within sorting complexity in external memory. The space bottleneck of the algorithm occurs when inducing suffixes during the second step. Suppose we respectively use a and b words to represent an input character and its type, then it takes at most $(3a+3b)n$ words for storing $arr_{\mathsf{L}}, arr_{\mathsf{S}}$ and arr_{LMS} and $(4+a+b)n$ words for sorting elements in $PQ_{\mathsf{L}}/PQ_{\mathsf{S}}$.

4 EXPERIMENTS

We implement the prototypes of Algorithms 1 and 3 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 2.4 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 [**?**] 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.

For description convenience, we denote the programs for Algorithms 1 and 3 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 3 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., [**?**], [**?**]) 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 3.

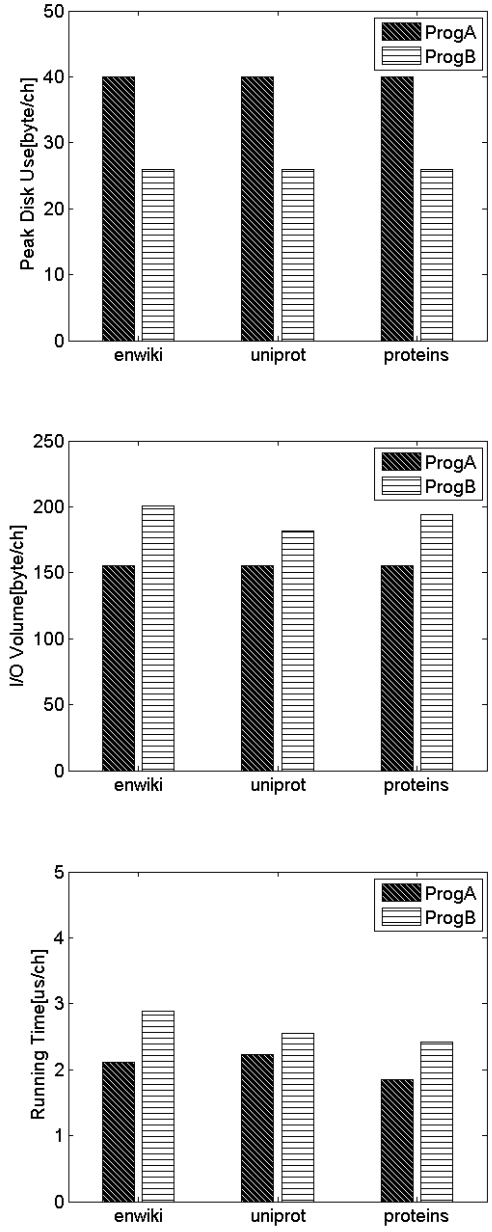

Fig. 1. Experimental results for various corpora.

Currently, for Algorithm 3, 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.

5 CONCLUSIONS

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

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.

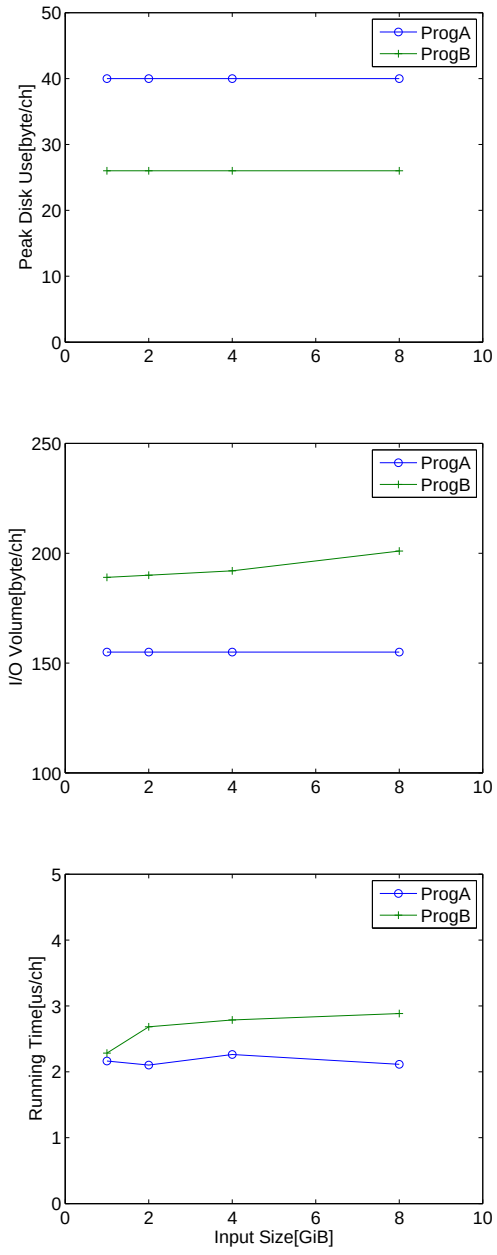

Fig. 2. Experimental results for prefixes of "enwiki".

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.