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 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 **??**,and give the conclusion in Section **??**.

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 **[?]** 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 **[?]** 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 **[?]** to design a probabilistic algorithm for computing a sparse suffix array.

Algorithm 1: The Algorithm for checking the conditions of 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 tuples in ST_1, ST_2 and ST_3 by 1st component.
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 **else**
11 **return** false // condition (1) is violated
12 **while** ST_2.notEmpty() and ST_2.top().$1st = i$ **do**
13 $e := ST_2$.top(), ST_2.pop(), $e.3rd := fp$, $e.4th := x[i]$, ST'_2.push(e)
14 **while** ST_3.notEmpty() and ST_3.top().$1st = i$ **do**
15 $e := ST_3$.top(), ST_3.pop(), $e.3rd := fp$, $e.4th := x[i]$, ST'_3.push(e)
16 $fp := fp \cdot \delta + x[i] \mod P$
17 sort tuples in ST_1, ST_2 and ST_3 by 2nd component.
18 **for** $i \in [1, n-1)$ **do**
19 $fp_1 := ST'_1$.top().$3rd$, ST'_1.pop(), $fp_2 := ST'_2$.top().$3rd$, $ch_1 := ST'_2$.top().$4th$, ST'_2.pop()
20 $\hat{fp_1} = fp2 - fp1 \cdot \delta^{lcp[i]} \mod P$
21 $fp_1 := ST'_1$.top().$3rd$, $fp_3 := ST'_2$.top().$3rd$, $ch_2 := ST'_3$.top().$4th$, ST'_3.pop()
22 $\hat{fp_2} = fp3 - fp1 \cdot \delta^{lcp[i]} \mod P$
23 **if** $\hat{fp_1} \neq \hat{fp_2}$ or $ch_1 \leq ch_2$ **then**
24 **return** false // condition (2) or (3) is violated
25 **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 6-16 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 17). 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 9-12). 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 20 and 22. 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 [**?**]. Besides, the algorithm reaches its peak disk use when sorting tuples in lines 5 and 17. 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

As will be seen from our experimental study in Section **??**, Algorithm 1 is quite space consuming, its peak disk use is 40 bytes per input character. In this section, we describe an alterative based on the induced-sorting principle, where the algorithm designed by this method only takes half space on real-world datasets in comparison with Algorithm 1.

3.1 Preliminaries

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

L-type/S-type/LMS character and suffix. 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.

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.

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

3.2 Idea

The induced sorting (IS) principle has been successfully used to design algorithms for building 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^*. In other words, suppose sa^* and lcp^* are already known, we can directly compute the suffix and LCP arrays by calling the inducing process of an existing IS-based construction algorithm. This enlightens us to use 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 procedure of an existing IS-based construction algorithm.

Similar to Section 2.2, we draw the conclusion in Corollary 2 by using the fingerprinting techinque to check condition (1) in Lemma 2.

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

3.3 Algorithm

The algorithm for checking Corollary 2 The following procedure can b For checking the conditions in 2, we

Algorithm 2: The Algorithm for checking sa^* and lcp^*.

Input: x, sa, lcp, n

ST_1 := $[(sa[i], i)|i \in [0, n)]$
sort tuples in ST_1 by 1st component
$j := 0$
for $i \in (n, 0]$ **do**
    $e := ST_1$.top(), ST_1.pop()
    if $x[i]$ *is S*-type* **then**
        $e.1st := j$, $j := j + 1$, ST_2.push(e)
sort tuples in ST_2 by 2nd component
$i := 0, j := 0, lcp_{min} := max\_val$ // use lcp_{min} to trace the LCP-value of two successive S*-type suffixes
while ST_2.NotEmpty() **do**
    $e := ST_2$.top(), ST_2.pop()
    while *true* **do**
        $i := i + 1$, $lcp_{min} := \mathsf{min}(lcp_{min}, lcp[i])$
        if $e.2nd = i$ **then**
            $sa^*[j] := e.1st$, $lcp^*[j] := lcp_{min}$, $j := j + 1$
            break
    $lcp_{min} := max\_val$
CheckByFP_Modified(x, sa^*, lcp^*, n_1)