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 fundamental data structures, and there are at least two needs in practice for checking their correctness, i.e. program debugging and verifying the arrays constructed by probabilistic algorithms. In this paper, we propose two probabilistic methods to check the suffix and LCP arrays of constant or integer alphabets in external memory by using a Karp-Rabin fingerprinting technique, where the checking result is wrong only with a negligible error probability. The first method checks the lexicographical order and the LCP-value of two suffixes by computing and comparing the fingerprints of their LCPs. This method is rather general in terms of that it can verify any full or sparse suffix/LCP array of any order. The second method is more space efficient, it first employs the fingerprinting technique to verify a subset of the given suffix and LCP arrays, from which then a copy of the suffix and LCP arrays is produced by using the induced sorting method and compared with the given arrays for verification, where the copy of the induced suffix and LCP arrays can be removed for constant alphabets.

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

1 Introduction

1.1 Background

Suffix and longest common prefix (LCP) arrays play an important role in various string processing tasks, such as data compression, pattern matching and genome assembly. In many applications, these two data structures make up the core part of a powerful full-text index, called enhanced suffix array [1], which is more space efficient than a suffix tree and applicable to emulating most searching functionalities provided by the latter in the same time complexity. The first algorithm for building suffix array (SA) in internal memory was presented in [2]. From then on, much more effort has been put on designing efficient SA constructors on different computation models [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. In respect of the research on LCP array construction algorithms, the existing works can be classified into two categories with regard to their input requirements, where the algorithms from the first category compute both suffix and LCP arrays at the same time with the original text only [10], [14], [15], and those from the second category carry out the computation by taking SA and/or Burrows-Wheeler transform (BWT) as additional inputs [14], [16], [17], [18], [19]. So far, the algorithms designed by the induced sorting (IS) method take linear time and space to run and get the best results on both internal and external memory models [6], [11], [20]. In addition to these sequential algorithms, there are also some parallel alternatives proposed to achieve high performance by fully using the available multi-core CPUs and/or GPUs [19], [21], [22], [23], [24].

While the research on efficient construction of suffix and LCP arrays keeps evolving, the algorithms proposed recently are becoming more complicated than before. Currently, the open source programs for the state-of-the-art algorithms are provided ”as-is” for demonstration and experiment purposes only, giving no guarantee that they have correctly implemented the algorithms. As a common practice, a suffix or LCP checker is provided to check the correctness of a constructed array. For example, such a checker can be found in some widespread software packages for DC3 [25], SA-IS [6], eSAIS [10] and so forth. In addition to help avoid implementation bugs, a checker is also demanded for an array constructed by a probabilistic algorithm (e.g. [26]). In this case, the array is correctly constructed with a probability and hence must be verified to ensure its correctness. As far as we know, the work in [27] describes the only SA checking method that can be found in the existing literature, and no efficient checking method for LCP array has been reported yet. Particularly, there is currently no reported solution that can check both the suffix and the LCP arrays in external memory. This motivates our work here to design efficient checkers for big suffix and LCP arrays.

1.2 Contribution

Our contribution comprises two methods to probabilistically verify any given suffix and LCP arrays. In principle, Method A checks the lexical order and the LCP-value of two neighboring suffixes in the given SA by literally comparing the characters of their LCPs. For reducing the time complexity of a comparison between two sequences of characters, we use a Karp-Rabin fingerprinting technique to convert each sequence into a single integer, called fingerprint, and compare the fingerprints instead to check the equality of

- *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.*

two sequences. The algorithm for Method A involves multiple scans and sorts on sets of $\mathcal{O}(n)$ fixed-size items. Its implementation in external memory suffers from a space bottleneck due to the large disk volume taken by each sort. To overcome this drawback, Method B first employs the fingerprinting technique to check a subset selected from the given suffix and LCP arrays, then it utilizes the IS method to induce the final suffix and LCP arrays from the verified subset and literally compares them with the input arrays to ensure the correctness of the latter. Our experimental results indicate that the program for Algorithm 2 designed by Method B only takes around half as much disk space as the program for Algorithm 1 designed by Method A.

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

2 METHOD A

2.1 Preliminaries

Given a string $x[0, n-1]$ drawn from a constant or integer alphabet Σ of size $O(1)$ or $O(n)$, respectively, 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 $i, j \in [0, n)$ and $i < j$, where $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$ are two suffixes starting with $x[sa[i]]$ and $x[sa[j]]$, respectively. Particularly, we say $\mathsf{suf}(sa[j])$ is a lexical neighbor of $\mathsf{suf}(sa[i])$ if $|i-j| = 1$. The LCP array of x, denoted by lcp, consists of n integers, where $lcp[0] = 0$ and $lcp[i]$ records the LCP-value of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[i-1])$ for $i \in [1, n)$.

2.2 Idea

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

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

(1) sa is a permutation of $\{0, 1, \ldots, n-1\}$.
(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. □

If we directly compare the characters of two suffixes, the worst case time is $O(n)$. An alternative is to exploit a perfect hash function to convert each substring into a single integer such that any two substrings have a common hash value if and only if they are literally equal to each other, hence we can compare the hash values of two substrings instead to check their equality. Taking into account the high difficulty of finding such a perfect hash function, we prefer using a Karp-Rabin fingerprinting function [28] to transform a substring into an integer called fingerprint. To be specific, suppose L is a prime and δ is a number randomly chosen from $[1, L)$, the fingerprint $\mathsf{fp}(i, j)$ for a substring $x[i, j]$ can be iteratively calculated according to the formulas below as follows: scan x rightward to iteratively compute $\mathsf{fp}(0, k)$ for all $k \in [0, n)$ using Formulas 1-2, record $\mathsf{fp}(0, i-1)$ and $\mathsf{fp}(0, j)$ during the calculation and subtract the former from the latter to obtain $\mathsf{fp}(i, j)$ using Formula 3.

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

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

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

Notice that two equal substrings always share a common fingerprint, but the inverse is not true. It has been proved in [28] that the probability of a false match can be reduced to a negligible level by setting L to a large value[1]. Hence, we have:

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) $x[sa[i] + lcp[i]] > x[sa[i-1] + lcp[i]]$.

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

2.3 Algorithm

We describe an algorithm for checking the conditions in Corollary 1 on random access models, of which the core part is to check the lexical order and the LCP-value for each pair of neighboring suffixes in sa on-the-fly during the scan of sa and lcp. This is done by using Formulas 1-3 following our discussion in the previous subsection. Two zero-initialized array, namely fp and mk, are introduced to facilitate the checking process, where fp is for storing the fingerprints of all the prefixes in x and mk is for checking whether or not each number of $\{0, 1, ..., n-1\}$ is present in sa.

1. This property is utilized in [26] to design a probabilistic algorithm for computing a sparse suffix array.

p:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[p]$:	2	1	3	1	3	1	2	1	3	1	3	1	2	1
$sa[p]$:	13	11	5	9	3	7	1	12	6	0	10	4	8	2
$lcp[p]$:	0	1	3	1	5	3	7	0	2	8	0	4	2	6

Compute $\mathrm{fp}(0, \mathrm{p})$ for $\mathrm{p} \in [0, \mathrm{n})$:

$\mathrm{fp}(0,0) = \mathrm{fp}(0,-1) \cdot 101 + x[0] \bmod 197 = 2,$

$\mathrm{fp}(0,1) = \mathrm{fp}(0,0) \cdot 101 + x[1] \bmod 197 = 6,$

$\mathrm{fp}(0,2) = \mathrm{fp}(0,1) \cdot 101 + x[2] \bmod 197 = 18,$

……

$\mathrm{fp}(0, \mathrm{p})$:	2	6	18	46	118	99	151	83	112	84	16	41	6	16

For suf(sa[0]) and suf(sa[1]):

$\mathrm{fp}(\mathrm{sa}[1], sa[1] + lcp[1] - 1) = fp(11) - fp(10) \cdot 101^1 \bmod 197$

$= 1$

$\mathrm{fp}(\mathrm{sa}[0], sa[0] + lcp[1] - 1) = fp(13) - fp(12) \cdot 101^1 \bmod 197$

$= 1$

For suf(sa[1]) and suf(sa[2]):

$\mathrm{fp}(\mathrm{sa}[2], sa[2] + lcp[2] - 1) = fp(7) - fp(4) \cdot 101^3 \bmod 197$

$= 160$

$\mathrm{fp}(\mathrm{sa}[1], sa[1] + lcp[2] - 1) = fp(13) - fp(10) \cdot 101^3 \bmod 197$

$= 160$

……

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

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

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 perform substeps (a)-(c) in sequence:

- (a) Retrieve $fp[u-1]$ and $fp[u+v-1]$ from fp to compute $\mathsf{fp}(u, u+v-1)$. Set $mk[u]$ to 1.
- (b) Retrieve $fp[w-1]$ and $fp[w+v-1]$ from fp to compute $\mathsf{fp}(w, w+v-1)$.
- (c) Check if $\mathsf{fp}(u, u+v-1) = \mathsf{fp}(w, w+v-1)$ and $x[u+v] > x[w+v]$.
- (d) Set $mk[sa[0]] = 1$.

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

It is clear that the above algorithm consumes $\mathcal{O}(n)$ time and space when implemented in internal memory. However, if the two auxiliary arrays cannot fit entirely in RAM during the execution of S2, it suffers from a performance degradation caused by frequent random accesses to disks. Assume that x, sa and lcp are stored in external memory, we design Algorithm 1 for conducting these I/O operations in a disk-friendly way. The main idea is to first sort data in their access order and then visit them sequentially. To the end, Algorithm 1 first scans sa and lcp to produce ST_1, ST_2, ST_3 and sorts their tuples by 1st component in ascending order at the very beginning (lines 2-5). Afterward, it iteratively computes the fingerprints of all the prefixes according to Formulas 1-2 and assigns them to the sorted tuples as following (lines 6-21): when figuring out $\mathsf{fp}(0, i-1)$, extract each tuple e with $e.1st = i$ from $ST_1/ST_2/ST_3$, update e with $\mathsf{fp}(0, i-1)$, and then forward e to $ST'_1/ST'_2/ST'_3$. Because the 1st components of the tuples in ST_1 constitute a copy of sa, the algorithm checks condition (1) when scanning these tuples in their sorted order. Finally, it sorts the updated tuples back to their original order (line 22) and visits them sequentially to check conditions (2)-(3) following the same way of S2 (lines 23-31).

The last point to be mentioned here is how to obtain the value of $\delta^{lcp[i]}$ quickly when computing $\hat{fp}_1$ and $\hat{fp}_2$ in lines 25 and 27. One method is to keep a lookup table in internal memory to store $\delta^{lcp[i]}$ for all $i \in [0, n)$. This can answer the question in constant time, but it is impractical if the lookup table exceeds the available memory capacity. Notice that the LCP of any two suffixes is smaller than n, thus we can return the answer in $\mathcal{O}(\lceil \log_2 n \rceil)$ time using $\mathcal{O}(\lceil \log_2 n \rceil)$ RAM space based on the following idea. Suppose e is an integer from $[0, n)$, it can be broken down into the form of $\Sigma_{i=0}^{\lceil \log_2 n \rceil} k_i \cdot 2^i$ by performing at most $\lceil \log_2 n \rceil$ divisions, where $k_0, k_1, ..., k_{\lceil \log_2 n \rceil}$ are also integers from $\{0, 1\}$. Thereby, we have $\delta^e = \Pi_{i=0}^{\lceil \log_2 n \rceil} \delta^{k_i \cdot 2^i}$ by replacing e with its decomposition, where the expression on the right side of the equation can be easily computed with $\{\delta^1, \delta^2, \ldots, \delta^{2^{\lceil \log_2 n \rceil}}\}$ already known.

2.4 Analysis

Algorithm 1 performs multiple scans and sorts on the arrays of $\mathcal{O}(n)$ fixed-size tuples residing on disks. Given an external memory model with RAM size M, disk size D and block size B, all are in words, the time and I/O complexities for each scan are $\mathcal{O}(n)$ and $\mathcal{O}(n/B)$, respectively, while those for each sort are $\mathcal{O}(n \log_{M/B}(n/B))$ and $\mathcal{O}((n/B) \log_{M/B}(n/B))$, respectively [29]. This algorithm reaches its peak disk use when sorting tuples in lines 5 and 22. Assume we represent each fingerprint by an α-byte integer, and encode the input string and the suffix/LCP array by β- and γ-byte integers, respectively, then it takes $(2 \cdot (\alpha + 2 \cdot \gamma)) \cdot n$ space for sorting ST_1/ST'_1 and $(2 \cdot (\alpha + \beta + 2 \cdot \gamma)) \cdot n$ for sorting ST_2/ST'_2 and ST_3/ST'_3. To improve space efficiency, our program implementing Algorithm 1 sorts ST_1/ST'_1, ST_2/ST'_2 and ST_3/ST'_3 separately and performs a single scan over x for each of them to obtain the target fingerprints, resulting in smaller space consumption but larger running time. However, from our experiments in Section 4, this program is still rather space consuming, its peak disk use is 40 times the size of x.

3 Method B

3.1 Preliminaries

In this part, we design another checking method using the fingerprinting and the induced sorting techniques. Before the presentation, we introduce some symbols and notations used in the following paragraphs.

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

Suffix and LCP buckets. Suppose sa is correct, all the suffixes in sa are naturally partitioned into multiple buckets and those of a common heading character are grouped into

Algorithm 1: The Algorithm Based on Corollary 1.

```
Function CheckByFP(x, sa, lcp, n)
    ST_1 := [(sa[i], i, null) | i ∈ [0, n)]
    ST_2 := [(sa[i] + lcp[i+1], i, null, null) | i ∈ [0, n−1)]
    ST_3 := [(sa[i] + lcp[i], i, null, null) | i ∈ [1, n)]
    sort tuples in ST_1, ST_2 and ST_3 by 1st component
    fp := 0
    for i ∈ [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 · δ + x[i] mod P                          // x[n] is the virtual character
    end
    sort tuples in ST'_1, ST'_2 and ST'_3 by 2nd component.
    for i ∈ [1, n) 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()
        f̂p_1 = fp_2 − fp_1 · δ^{lcp[i]} mod P
        fp_1 := ST'_1.top().3rd, fp_3 := ST'_3.top().3rd, ch_2 := ST'_3.top().4th, ST'_3.pop()
        f̂p_2 = fp_3 − fp_1 · δ^{lcp[i]} mod P
        if f̂p_1 ≠ f̂p_2 or ch_1 ≥ ch_2 then
            return false                                   // condition (2) or (3) is violated
        end
    end
    return true
```

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

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

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

3.2 Idea

An IS suffix and LCP arrays construction algorithm mainly consists of a reduction phase for computing sa^* and lcp^*, followed by an induction phase for inducing sa and lcp from sa^* and lcp^* [2]. This enlightens us to design a checker based on Lemma 2, provided that sa^* and lcp^* are already known.

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 the IS method.

Similar to Method A, the fingerprinting technique can be employed to probabilistically check the correctness of sa^* and lcp^*. An external-memory algorithm for checking the conditions in Corollary 2 is given in the next subsection.

Corollary 2. Both $sa[0, n)$ and $lcp[0, n)$ are correct with a high probability given the following conditions:

(1) $x[sa^*[i]]$ is S*-type and $sa^*[j] \neq sa^*[i]$ for $i, j \in [0, n_1)$ and $i \neq j$.
(2) $\mathsf{fp}(sa^*[i], sa^*[i] + lcp^*[i] - 1) = \mathsf{fp}(sa^*[i-1], sa^*[i-1] + lcp^*[i] - 1)$ for $i \in [1, n_1)$.
(3) $x[sa^*[i] + lcp^*[i]] > x[sa^*[i-1] + lcp^*[i]]$ for $i \in [1, n_1)$.
(4) $sa[i] = sa'[i]$ and $lcp[i] = lcp'[i]$ for $i \in [0, n)$, where sa' and lcp' are induced from sa^* and lcp^* by the IS method.

2. In the interest of completeness, we give an overview of the induction phase in Appendix A.

3.3 Algorithm

The first step of Algorithm 2 is to compute and verify sa^* and lcp^*. According to their definitions, sa^* can be produced by sequentially retrieving the S*-type suffixes from sa, while the LCP-value of two successive S*-type suffixes in sa, say $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$, is equal to the minimal of $\{lcp[i+1], ..., lcp[j-1], lcp[j]\}$. Hence, the proposed algorithm first sorts all the suffixes in sa by their starting positions (lines 2-3) and then scans x only once to pick out the S*-type suffixes (lines 4-13). After that, it puts these S*-type suffixes back in their lexical order and outputs them one by one to generate sa^* (lines 14-27). Meanwhile, it calculates the LCP-value for each pair of the neighboring suffixes in sa^* by tracing the minimal over the lcp interval indicated by the two suffixes. Notice that, we check condition (1) in Corollary 2 during the time when visiting the suffixes in position order (lines 7-12) and conditions (2)-(3) by calling Algorithm 1 with sa^* and lcp^* as inputs. If sa^* and lcp^* are both correct, then Algorithm 2 invokes an inducing process to employ IS method for inducing sa' and lcp' from the two verified arrays (line 31) and literally compares them with sa and lcp to complete the whole checking process (lines 32-36).

Assume that the alphabet Σ is of a constant size, we can check sa and lcp without producing a copy of the final suffix and LCP arrays in Algorithm 2. The idea is to compare the induced suffix/LCP items with their corresponding items in sa/lcp during the inducing process. Specifically, when a suffix/LCP item v_1 is induced into a bucket, we check if it is equal to the corresponding item v_2 in sa/lcp. If $v_1 = v_2$, then v_2 is correct and we further use this value to induce the remaining suffix/LCP items. We describe more details of the adapted inducing process as below, where lp_1/lp_2 and sp_1/sp_2 point to the next items to be visited in the L-type and S-type sub-buckets, respectively.

S1 (a) Let $lp_1[c]$ and $lp_2[c]$ point to the leftmost items of $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{lcp\_bkt_L}(c)$, for $c \in [0, \Sigma)$.
   (b) Scan sa and lcp rightward to induce L-type suffixes and their LCP-values. For each induced suffix p (with a heading character c_0) and its LCP-value q: (1) check if $p = lp_1[c_0]$ and $q = lp_2[c_0]$; (2) move $lp_1[c_0]$ and $lp_2[c_0]$ to the next items on the right.

S2 (a) Let $sp_1[c]$ and $sp_2[c]$ point to the rightmost items of $\mathsf{sa\_bkt_S}(c)$ and $\mathsf{lcp\_bkt_S}(c)$, for $c \in [0, \Sigma)$.
   (b) Scan sa and lcp leftward to induce S-type suffixes and their LCP-values. For each induced suffix p (with a heading character c_0) and its LCP-value q: (1) check if $p = sp_1[c_0]$ and $q = sp_2[c_0]$; (2) move $sp_1[c_0]$ and $sp_2[c_0]$ to the next items on the left.

3.4 Analysis

Algorithm 2 consists of two steps. The first step for checking sa^* and lcp^* can be done in sorting complexity, while the second step for checking sa and lcp can also be done in sorting complexity for inducing the copy of suffix and LCP arrays by using external-memory priority queues. The experimental results in Section 4 indicate that the disk use of the programs for this algorithm and its tuned version are around $21n$ and $26n$, respectively, where the peak values are achieved when executing the second step.

4 Experiments

4.1 Setup

For engineering simplicity, the algorithms proposed in the previous sections are implemented using the external-memory containers from the STXXL library [30]. We make a performance evaluation by running programs on the real-world corpora listed in Table 1, where three measures normalized by the size of the input string are investigated:

- RT: running time, in microseconds.
- PDU: peak disk use of external memory, in bytes.
- IOV: amount of data read from and write to external memory, in bytes. Each element of the suffix and LCP arrays takes 5 bytes.

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

4.2 Results

Fig. 2 illustrates the performance comparison of ProgA and ProgB on different datasets, where "enwiki_8g" consists of the leftmost 8 GiB extracted from "enwiki". As depicted, ProgB runs slower than ProgA by around 20%. The speed gap is mainly due to the difference in I/O performance. Specifically, the I/O volume of ProgA keeps at $155n$ for all the three datasets, while that of ProgB rises up to nearly $200n$ on average. Besides, the PDU of ProgB is about $26/40 = 0.65$ as ProgA. Recall that Algorithm 2 invokes Algorithm 1 to check the suffix and LCP arrays for the S*-type suffixes. Because at most one out of every two successive characters in the input string is S*-type, the consumption for checking the suffix and LCP arrays of S*-type suffixes in ProgB is expected to be half as that for checking the given arrays in ProgA. For a better insight, we collect in Table 2 the performance overhead of ProgB and ProgA when checking the suffix and LCP arrays of S*-type suffixes and all, respectively. As shown in the table, the mean ratio of the number of S*-type suffixes to the number of all the suffixes is around 0.30 for the datasets under investigation, while the mean ratios of time, space and I/O volume for checking sa^* and lcp^* to that for checking sa and lcp are 0.38, 0.57 and 0.60, respectively.

The above observations indicate that ProgB reaches its peak disk use when checking the final suffix and LCP arrays during the inducing process, i.e., the inducing process constitutes the performance bottleneck of the whole algorithm. By adopting the space optimization scheme introduced in Section 3.3, we adapt Algorithm 2 and evaluate the tuned version of ProgB, called ProgB+, in comparison with ProgA and ProgB. Fig. 2 shows that the maximum space requirement for ProgB+ is about $21n$, which is much less than that of ProgB and only half as that of ProgA. In addition, progB+ outperforms its prototype with respect to time and I/O efficiencies and is even faster than ProgA when handling "uniprot". We also investigate the performance trend of the three programs on the prefix of "enwiki" with the length

Algorithm 2: The Algorithm Based on Corollary 2.

```
Function CheckByIS(x, sa, lcp, n)
    ST_1 := [(sa[i], i, null) | i ∈ [0, n)]
    sort tuples in ST_1 by 1st component
    pos := −1
    for i ∈ (n, 0] do
        e := ST_1.top(), ST_1.pop()
        if x[i] is S*-type then
            if pos ≥ e.1st then
                return false            // condition (1) is violated
            end
            ST_2.push(e), pos := e.1st
        end
    end
    sort tuples in ST_2 by the 2nd component
    i := 0, j := 0, lcp_min := max_val
    while ST_2.NotEmpty() do
        e := ST_2.top(), ST_2.pop()
        while true do
            lcp_min := min(lcp_min, lcp[i])
            if e.2nd = i then
                sa*[j] := e.1st, lcp*[j] := lcp_min, j := j + 1, i := i + 1
                break
            end
            i := i + 1
        end
        lcp_min := max_val
    end
    if CheckByFP(x, sa*, lcp*, n_1) = false then
        return false            // conditions (2) or (3) is violated
    end
    (sa', lcp') := InducingProcess(x, sa*, lcp*)
    for i ∈ [0, n) do
        if sa[i] ≠ sa'[i] || lcp[i] ≠ lcp'[i] then
            return false            // condition (4) is violated
        end
    end
    return true
```

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

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

varying in $\{1, 2, 4, 8\}$ GiB. In Figure 3, the peak disk use for each program remains unchanged, but their speed become slower as the prefix length increases due to the performance degradation of the external memory sorter exploited in our programs. This can be also observed from Table 2, where ProgB+ keeps the I/O volume around $90n$ with the prefix length of "enwiki" varying from 1 to 8 GiB but its running time rises from 1.05 to 1.33.

In the next experiment, we compare our programs with two solutions for building the suffix and LCP arrays as below, where each of them combines an existing suffix sorter with an LCP builder:

- Solution 1: Use eSAIS for building SA and the sequential version of Sparse-ϕ [24] for building LCP array.
- Solution 2: Use pSAscan [23] for building SA and the parallel version of Sparse-ϕ for building LCP array.

We select these solutions because they are the current fastest available to us. A runtime breakdown of these solutions on the prefixes of "enwiki" is given in Table 3, from which we see that Solution 2 is about two times faster than Solution 1 and twice as fast as ProgA. This phenomenon is mainly due to the high speed of pSAscan in the experiment. However, it is worthy of pointing out that both pSAscan and

TABLE 2
A performance comparison of checking the suffix and LCP arrays of S*-type suffixes to checking that of all the suffixes.

Dataset	# of suffixes			PDU			IOV			RT		
	S*-type	all	ratio	S*-type	all	ratio	S*-type	all	ratio	S*-type	all	ratio
enwiki_1g	329810376	1073741824	0.31	15.67	40	0.39	89.94	155	0.58	1.05	1.70	0.62
enwiki_2g	650901939	2147483648	0.30	15.41	40	0.39	89.18	155	0.58	1.22	1.85	0.66
enwiki_4g	1301327878	4294967296	0.30	15.45	40	0.39	89.14	155	0.58	1.19	1.89	0.63
enwiki_8g	2586471839	8589934592	0.30	15.35	40	0.38	88.80	155	0.57	1.33	2.14	0.62
uniprot	829262945	3028811776	0.27	13.94	40	0.35	83.80	155	0.54	1.04	2.26	0.46
proteins	379092002	1184366592	0.32	16.21	40	0.41	92.29	155	0.60	1.14	1.85	0.62
mean	1012811163	3189156522	0.30	15.34	40	0.38	88.86	155	0.57	1.16	1.95	0.60

Sparse-ϕ are of the time and I/O complexities proportional to n^2/M. This is much higher than eSAIS and our checking algorithms when n increases, and thus poses a strict limitation to the scalability of Solution 2. As reported in [23], when n is considerably greater than M, eSAIS is much more time and I/O efficient than pSAscan. In this experiment, pSAscan builds the SA for "enwiki_8g" in time double as that for "enwiki_1g". For big n, it is more reasonable to compare the results of our programs with that of Solution 1.

4.3 Discussion

There are still several ways to enhance the performance of our programs. Firstly, it was observed that our programs suffer from a bottleneck when sorting massive data in external memory. For implementation simplicity, we currently use the container provided by the STXXL library to execute the sorting task without designing a specific sorter optimized for our purpose. It is possible to speed up the sorting process by high-performance radix-sort GPU algorithms. Secondly, Method B checks the suffix and LCP arrays using the IS method. At the time of writing this paper, the existing IS suffix/LCP array construction algorithms are naturally sequential. We have been conducting a study to design IS parallel algorithms, this work may also improve the implementation design of Algorithm 2.

In this paper, both methods A and B assume a constant or integer alphabet. However, in practice, an input string is commonly of a constant alphabet, e.g. 4 and 256 characters for genome and text, respectively. In this case, Method B can be improved for better time and space performance by inducing the final suffix and LCP arrays directly from sa_S/lcp_S or sa_L/lcp_L, which consist of all the sorted S-type or L-type suffixes with their LCP-values and can be obtained as follows. Given the alphabet is constant, we first scan the input string once to get the statistics for buckets in the input suffix array. Without loss of generality, suppose that the S-type characters are less, then we scan the suffix array once to get sa_S/lcp_S by using the bucket statistics to on-the-fly determine a scanned suffix is S-type or not. Afterward, we check sa_S/lcp_S by using Algorithm 1 and induce sa'/lcp' from them. In this way, we avoid the two integer sorts in the current fashion of Method B for retrieving sa^*/lcp^* and speed up the inducing process by nearly half as well.

It should be noticed that our programs are coded for experimental study only. From engineering aspects, there is still a big margin for better implementation. For example, several algorithms for induced sorting a suffix and/or LCP arrays were proposed these years [10], [12], [13], with different methods for solving the key problem of retrieving the preceding character of a sorted suffix in the inducing process. A recent work [20] for engineering these IS methods with some implementation optimizing techniques achieves a significant improvement over the previous results. As reported, the peak disk use is $7n$ for 32-bit integers. Because the inducing process is the performance bottleneck for Method B, it is reasonable to expect that a better engineering implementation of the method will yield a remarkable performance improvement. An optimized engineering of our methods is out of the scope of this paper and will be addressed elsewhere.

Finally, we emphasize that Method A is able to check any set containing one or multiple pairs of lexicographically neighboring or non-neighboring suffixes. This feature can be applied to various scenarios. For example, a suffix/LCP array may be broken due to software or hardware malfunctions. If a backup is not available and it is too time-consuming to rebuild the whole array, then we can locate the bad areas quickly using Algorithm 1 and restore the partial SA for each area by calling a sparse SA construction algorithm. Another example is to check the correctness of a sparse SA. Because the number of suffixes in a sparse SA is commonly much smaller than that in the full SA, Algorithm 1 could become an efficient verification solution in this case.

5 Conclusions

In this paper, we propose two methods for probabilistically checking the given suffix and LCP arrays. Theoretically, the external-memory algorithms designed by these methods have better time and I/O complexities compared to the existing fastest construction algorithms. From our experiments, the current program for Algorithm 2 runs slower than that for Algorithm 1, but the former is more space efficient than the latter. As discussed in Section 4, our experimental programs can be further optimized to achieve better performance, especially when checking arrays of constant alphabets that are most common in practice.

From our perspective, a checker should be not only fast but also general. In practice, we usually check a constructed array from a builder by comparing it with that from another builder. But this is not feasible in all the cases, for example,

TABLE 3
A runtime comparison for the programs of two construction solutions and ours.

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

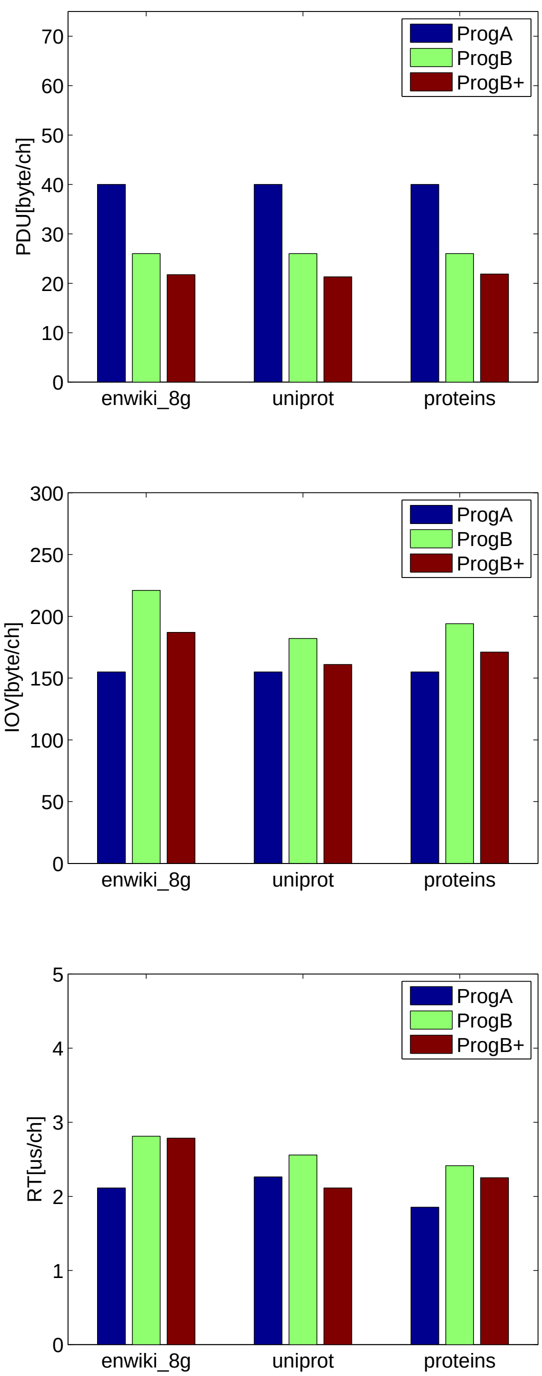

Fig. 2. Performance of ProgA, ProgB and ProgB+ for different corpora.

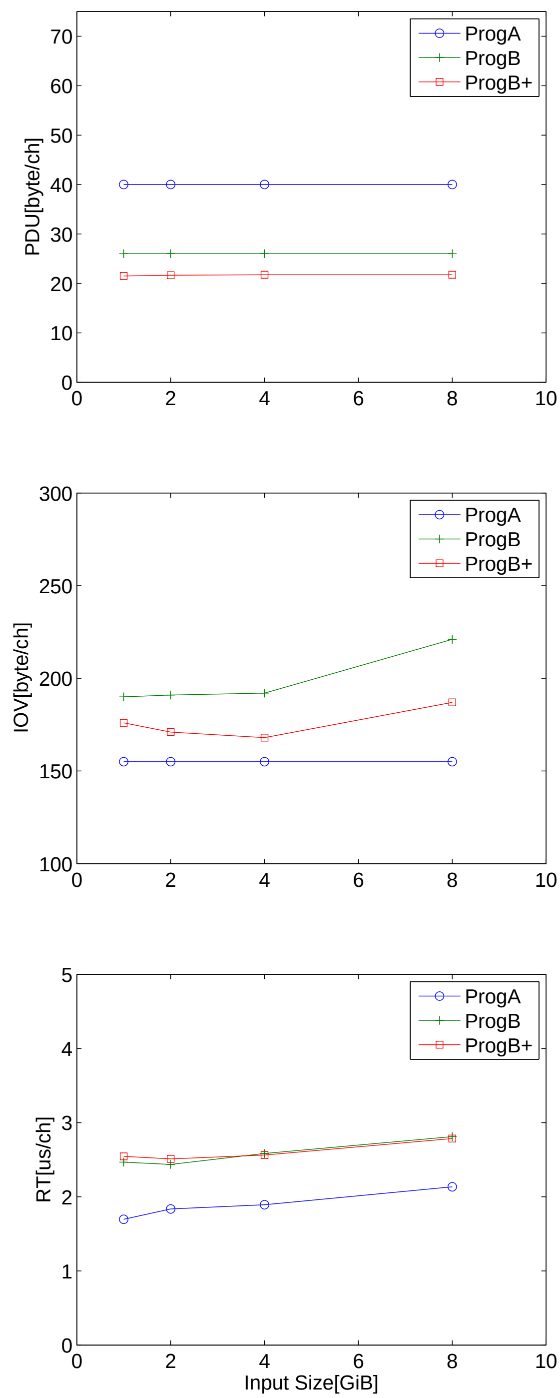

Fig. 3. Performance of ProgA, ProgB and ProgB+ for prefixes of "enwiki".

an algorithm for constructing infinite-order[3] array can not be directly used to check a finite-order array and vise versa. On the contrary, Method A can be generalized to check the correctness of the lexical order and the LCP-values of any pairs of suffixes. This makes it possible for verifying any full or sparse suffix/LCP array of any order.

The IS method has been employed to successfully design a number of suffix and LCP arrays construction algorithms. A recent work [20] reports that a careful engineering of the IS in external memory can build a suffix array using only $7n$ bytes for $n \leq 2^{32}$, which is approaching the optimal in respect to $5n$ bytes for the IS in internal memory. Besides, it runs fastest in most cases of the experiments therein. This convinces that the IS method could serve as a basis for developing potentially optimal solutions for building suffix/LCP arrays. We design here the algorithms for checking given suffix and LCP arrays. In another paper, we will come up with a solution for building and checking a suffix/LCP array simultaneously using the IS method. By this way, no additional checker is needed to be distributed with a suffix/LCP array builder using the IS method.

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.

3. A suffix array is infinite-order if the suffixes are sorted up to their ends, or else finite-order.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] J. Kärkkäinen, D. Kempa, S. J. Puglisi, and et al., "Engineering External Memory Induced Suffix Sorting," in *In Proceedings of the 19th Workshop on Algorithm Engineering and Experiments*, 2017, pp. 98–108.

[21] 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.

[22] 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.

[23] 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.

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

[25] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better External Memory Suffix Array Construction," *ACM Journal of Experimental Algorithmics*, vol. 12, pp. 3.4:1–3.4:24, August 2008.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

Appendix A
Overview on the Induction Phase

Notice that $\mathsf{suf}(i)$ is lexicographically smaller than $\mathsf{suf}(j)$ if and only if (1) $x[i] < x[j]$ or (2) $x[i] = x[j]$ and $\mathsf{suf}(i+1) < \mathsf{suf}(j+1)$. This property constitutes the core part of the IS method and has been utilized by SA-IS and other IS variants to derive the order of unsorted suffixes from the order of sorted ones following the 3-step induction phase below.

S1 Clear S-type sub-buckets in sa. Scan sa^* leftward and insert each element into the current rightmost empty position in the corresponding S-type sub-bucket.

S2 Clear L-type sub-buckets in sa and insert $n-1$ into the leftmost position in $\mathsf{sa\_bkt_L}(x[n-1])$. Scan sa rightward with i increasing from 0 to $n-1$. For each scanned non-empty $sa[i]$ with $t[sa[i]-1] = 0$, insert $sa[i]-1$ into the current leftmost empty position in $\mathsf{sa\_bkt_L}(x[sa[i]-1])$.

S3 Clear S-type sub-buckets in sa. Scan sa leftward with i decreasing from $n-1$ to 0. For each scanned non-empty $sa[i]$ with $t[sa[i]-1] = 1$, insert $sa[i]-1$ into the current rightmost empty position in $\mathsf{sa\_bkt_S}(x[sa[i]-1])$.

In brief, given sa^*, S1 inserts all the S*-type suffixes into sa in their lexical order. Then, S2-S3 induce the order of L- and S-type suffixes from those already sorted in sa, respectively, where the relative order of two suffixes induced into the same sub-bucket matches their insertion order according to the previously stated property.

We show in Fig. 4 a running example with more details. As depicted, the input string x contains 6 S*-type suffixes sorted in line 3. When finished inserting the S*-type suffixes in lines 5-6, we first find the head of each L-type sub-bucket (marked by the symbol $\wedge$) and insert $\mathsf{suf}(13)$ into sa. Notice that $\mathsf{suf}(13)$ is a single character, it must be the smallest L-type suffixes starting with 1. Thus, we put $\mathsf{suf}(13)$ into the leftmost position in $\mathsf{sa\_bkt_L}(1)$ in line 8. Then, we scan sa from left to right for inducing the order of all the L-type suffixes. In lines 10-11, when visiting $sa[0] = 13$ (marked by the symbol @), we check the type array t to find $x[12] = 2$ is L-type and hence insert $\mathsf{suf}(12)$ into the current leftmost empty position in $\mathsf{sa\_bkt_L}(2)$. Similarly, in lines 12-13, we visit the next scanned item $sa[1] = 11$ and see that $t[10] = 0$, thus we place $\mathsf{suf}(10)$ into the current head of $\mathsf{sa\_bkt_L}(3)$. Following this way, we get all the L-type suffixes sorted in sa. After that, we first find the end of each S-type sub-bucket in lines 25-26 and scan sa leftward for inducing the order of all the S-type suffixes in lines 27-40. When visiting $sa[13] = 2$, we see $x[1]$ is S-type and thus put $\mathsf{suf}(1)$ into the current rightmost empty position in $\mathsf{sa\_bkt_S}(1)$. Then, at $sa[12] = 8$, we see $x[7] = 1$ is S-type and thus put $\mathsf{suf}(7)$ into the current rightmost empty position in $\mathsf{sa\_bkt_S}(1)$. To repeat scanning sa in this way, we get all the S-type suffixes sorted in sa.

The work presented in [14] describes how to compute the LCP array during S2-S3 of the induction phase. Specifically, for any two suffixes placed at the neighboring positions in sa, their LCP-value can be computed according to one of the following two cases in respect to whether or not they are inserted into the same sub-bucket: if yes, then the LCP-value is one greater than that of the two suffixes from which inducing them; otherwise, the LCP-value is zero. The key operation herein is to compute the LCP-value

00	p:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
01	$x[p]$:	2	1	3	1	3	1	2	1	3	1	3	1	2	1
02	$t[p]$:	L	S*	L	S*	L	S*	L	S*	L	S*	L	S*	L	L
03	$sa^*[p]$:	11	5	9	3	7	1								
04	Insert the sorted S*-type suffixes into sa^*:														
05	bucket:				1					2			3		
06	$sa^*[p]$:	{-1	11	5	9	3	7	1}	{-1	-1	-1}	{-1	-1	-1	-1}
07	Sort L-type suffixes:														
08	$sa^*[p]$:	{13	11	5	9	3	7	1}	{-1	-1	-1}	{-1	-1	-1	-1}
09		^							^			^			
10		{13	11	5	9	3	7	1}	{12	-1	-1}	{-1	-1	-1	-1}
11		@^								^		^			
12		{13	11	5	9	3	7	1}	{12	-1	-1}	{10	-1	-1	-1}
13		^	@							^			^		
14		{13	11	5	9	3	7	1}	{12	-1	-1}	{10	4	-1	-1}
15		^		@						^				^	
16		{13	11	5	9	3	7	1}	{12	-1	-1}	{10	4	8	-1}
17		^			@					^					^
18		{13	11	5	9	3	7	1}	{12	-1	-1}	{10	4	8	2}
19		^				@				^					^
20		{13	11	5	9	3	7	1}	{12	6	-1}	{10	4	8	2}
21		^					@				^				^
22		{13	11	5	9	3	7	1}	{12	6	0}	{10	4	8	2}
23		^						@			^				^
24	Sort S-type Suffixes:														
25		{13	-1	-1	-1	-1	-1	-1}	{12	6	0}	{10	4	8	2}
26								^			^				^
27		{13	-1	-1	-1	-1	-1	1}	{12	6	0}	{10	4	8	2}
28							^				^				@^
29		{13	-1	-1	-1	-1	7	1}	{12	6	0}	{10	4	8	2}
30						^					^			@	^
31		{13	-1	-1	-1	3	7	1}	{12	6	0}	{10	4	8	2}
32					^						^		@		^
33		{13	-1	-1	9	3	7	1}	{12	6	0}	{10	4	8	2}
34				^							^	@			^
35		{13	-1	-1	9	3	7	1}	{12	6	0}	{10	4	8	2}
36				^							@^				^
37		{13	-1	5	9	3	7	1}	{12	6	0}	{10	4	8	2}
38			^							@	^				^
39		{13	11	5	9	3	7	1}	{12	6	0}	{10	4	8	2}
40		^							@		^				^

Fig. 4. An Example for inducing the suffix and LCP arrays.

of the inducing suffixes at the same time when inserting the induced suffixes into sa. For example, in lines 10-21 of Fig. 4, we induce $\mathsf{suf}(12)$ and $\mathsf{suf}(6)$ into the neighboring positions of $\mathsf{sa\_bkt_L}(2)$, respectively. If we keep recording the minimal over $lcp(0, 5]$, then we can obtain the LCP-value of the inducing suffixes $\mathsf{suf}(13)$ and $\mathsf{suf}(7)$ immediately after putting $\mathsf{suf}(6)$ into sa. This problem is modeled as a range minimum query in [14] and can be answered within amortized $\mathcal{O}(1)$ time.