Building and Checking a Suffix Array by Induced Sorting

Yi Wu, Ge Nong, Wai Hong Chan, and Bin Lao

Abstract—A suffix array can be built by the induced sorting (IS) method on both internal and external memory models. We propose two methods for checking a suffix array when it is being built by any IS suffix sorter, i.e. build and check a suffix array by the IS method simultaneously. In particular, the first method can be extended to check both suffix and LCP arrays. By combining the Karp-Rabin fingerprinting technique with our methods, we design two algorithms that perform checking correctly with a negligible error probability. The algorithm designed by the first method has an integer sorting complexity, while the algorithm designed by the second method runs in linear time. We integrate the algorithm designed by the second method into the existing disk-based suffix sorting algorithm DSA-IS and implement it for performance analysis. From our experiments, the checking overhead is considerably smaller than the building consumption.

Index Terms—Suffix and LCP arrays, construction and verification, internal and external memory.

In the accepted paper, the LCP-value ℓ of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ is assumed to be right if $\mathsf{sub}(i, i+\ell-1) = \mathsf{sub}(j, j+\ell-1)$ and $x[i+\ell] \neq x[j+\ell]$. Given that ℓ is correct, the lexical order of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ is checked by comparing $x[i+\ell]$ and $x[j+\ell]$.

In this paper, both the lexical order and the LCP-value of two neighboring suffixes in the given suffix array is checked at the top recursion level following the IS principle.

1 Introduction

A suffix array (SA) can be built within linear time and space by the internal-memory algorithm SA-IS [1] using the induced sorting method. According to the IS principle, the order of two suffixes is determined by comparing their heading characters and successors in sequence, where the order of their successors is determined recursively. Recently, the IS method has been also applied to designing three disk-based suffix sorting algorithms eSAIS [2], DSA-IS [3] and SAIS-PQ [4], where eSAIS can build both suffix and LCP arrays at the same time. Compared to the others designed by different methods (e.g., DC3 [5], bwt-disk [6], SAscan [7] and pSAscan [8]), these algorithms runs fast with linear I/O complexity. However, these algorithms currently suffer from a bottleneck due to the large disk space for obtaining the heading characters of unsorted suffixes and the ranks of their sorted successors in a disk-friendly way. It is reported that the average disk use for constructing an SA encoded by 40-bit integers for pSAscan is only $7.5n$ bytes, while that for eSAIS, DSA-IS and SAIS-PQ are $24n$, $18n$ and $15n$ bytes, respectively, where n is the size of input string. The poor space performance observed from these IS algorithms is mainly because the current programs for these algorithms fail to immediately free the disk space for temporary data even when the data is no longer needed. A dramatic improvement can be achieved by storing temporary data in multiple files and deleting a file when the data in the file is not needed any more. This trick has been used by fSAIS [9] to engineer the external-memory IS method for better performance, where the peak disk use is $8n$ bytes for constructing an SA of 40-bit integers in the given experiments.

While the research on developing various SA builders using the IS method is involving, the developed software are becoming complex and appear as open-source without guarantee. A constructed SA should be checked to detect potential errors caused by implementation bugs and other malfunctions. The software packages for DC3 and eSAIS provide a checker based on the idea presented in [5]. When running on external-memory model, the cost taken by this checker is rather high as it performs two passes of integer sorts for arranging $\mathcal{O}(n)$ fixed-size tuples. In this paper, we propose two methods to enable the IS method to build and check an SA simultaneously. In particular, the first method can be extended to check both suffix and LCP arrays. We employ the Karp-Rabin fingerprinting technique [10] to design two probabilistic algorithms, in terms of that their checking results are wrong with a negligible probability. For analysis, we first use new substring sorting and naming methods to improve the design of DSA-IS and then combine the second checking method into the adapted DSA-IS for evaluating the checking overhead. Our experimental results indicate that the time, space and I/O volume for checking SA is considerably smaller than that for building SA.

The rest of this paper is organized as follows. Section 2 introduces some notations and symbols for presentation convenience. Section 3 gives an overview of the existing IS suffix sorting algorithms and show the details of our new substring sorting and naming methods specific for DSA-IS. Section 4 presents the proposed checking methods and the

- *Y. Wu, G. Nong (corresponding author) and B. Lao 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, Laobin@mail3.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.*

probabilistic algorithms designed by them. Sections 5 and 6 show the experimental results and the concluding remarks, respectively.

2 PRELIMINARIES

Given a string $x[0, n)$ drawn from a full-ordered alphabet Σ, we assume the ending character $x[n-1]$ to be unique and lexicographically smaller than any other characters in x. For convenience, we denote by $\mathsf{suf}(i)$ and $\mathsf{sub}(i, j)$ the suffix running from $x[i]$ to $x[n-1]$ and the substring running from $x[i]$ to $x[j]$, respectively. The following notations are also used in our presentation.

Characters in x are classified into two categories: L-type and S-type. We say $x[i]$ is S-type if (1) $i = n-1$ or (2) $x[i] = x[i+1]$ and $x[i+1]$ is S-type; otherwise $x[i]$ is L-type. Further, if $x[i]$ and $x[i-1]$ are respectively S-type/L-type and L-type/S-type, then $x[i]$ is also called S*-type/L*-type. We use an array t to record the type of all the characters in x, where $t[i] = 1$ or 0 if $x[i]$ is S-type or L-type, respectively. The type of a substring or suffix is determined by that of its heading character.

Given two characters $x[i]$ and $x[i+1]$, we say $x[i]$ is the predecessor of $x[i+1]$ and $x[i+1]$ is the successor of $x[i]$. We define the predecessor-successor relationship between $\mathsf{suf}(i)/\mathsf{sub}(i, j)$ and $\mathsf{suf}(i+1)/\mathsf{sub}(i+1, j)$.

Partition x into multiple S*-type substrings, each substring contains two S*-type characters and any two neighboring substrings overlap an S*-type character. We produce a reduced string x_1 by replacing each substring with its name, where the name indicates the rank of the substring among all. In this paper, we use "rank" and "name" interchangeably to represent the lexical order of substrings.

The suffix array sa arranges all the suffixes of x in their lexical order, where $sa[i]$ records the starting position of the $(i+1)$-th smallest suffix. We also define the suffix array sa_1 for the reduced string x_1 in the same way.

The LCP array lcp records the LCP-value of each pair of neighboring suffixes in sa. We assume $lcp[0] = 0$ and let $lcp[i] = \ell$ for $i \in [1, n)$, where ℓ is the LCP-value of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[i-1])$.

All the suffixes in sa are naturally grouped into multiple buckets. Each bucket occupies a contiguous interval in sa and contains all the suffixes with a same heading character. Without loss of generality, we denote by $\mathsf{sa\_bkt}(c_0)$ the bucket for suffixes starting with c_0, where $\mathsf{sa\_bkt}(c_0)$ can be naturally divided into two parts $\mathsf{sa\_bkt_L}(c_0)$ and $\mathsf{sa\_bkt_S}(c_0)$ containing L-type and S-type suffixes, respectively. We also define $\mathsf{lcp\_bkt}(c_0)$, $\mathsf{lcp\_bkt_L}(c_0)$ and $\mathsf{lcp\_bkt_S}(c_0)$ on lcp for $c_0 \in \Sigma$.

Let $n_1 = \|x_1\|$, we use $sa^*[0, n_1)$ and $lcp^*[0, n_1)$ to denote the suffix and LCP arrays for S*-type suffixes in x, respectively. Specifically, $sa^*[i]$ records the starting position of the $(i+1)$-th smallest S*-type substring, while $lcp^*[i]$ records the LCP-value of $\mathsf{suf}(sa^*[i])$ and $\mathsf{suf}(sa^*[i-1])$.

3 BUILDER

Algorithm 1: The Framework of an IS suffix sorting algorithm.

Input: x
Output: sa
1 /* Reduction Phase */
2 Sort S*-type substrings by the IS method.
3 Name the sorted S*-type substrings to produce x_1.
4
5 /* Check Recursion Condition */
6 **if** *exist two equal characters in* x_1 **then**
7 Recursively call the reduction phase on x_1.
8 **end**
9 **else**
10 Compute sa_1 from x_1.
11 **end**
12
13 /* Induction Phase */
14 Sort suffixes by the IS method.

3.1 Introduction to IS Suffix Sorting Algorithms

Algorithm 1 shows the framework of an IS suffix sorting algorithm. In lines 2-3, a reduction phase for sorting and naming S*-type substrings is executed to produce the reduced string x_1. If there exist duplicate characters in x_1, then the reduction phase is recursively called with x_1 as input in line 7 and executed at the higher recursion level; otherwise, the S*-type suffixes in x are already sorted and sa_1 is directly computed from x_1 in line 10. Afterward, an induction phase for sorting suffixes is executed to produce sa at the current recursion level in line 14. The induction phase is called recursively with sa as input and executed at the lower recursion level until the final sa is generated.

During the execution of a reduction/induction phase, all the substrings/suffixes are sorted by comparing their heading characters and the ranks of their successors according to the IS principle. This involves a great number of random accesses to x and sa, which can be done very fast if both x and sa can wholly reside on RAM. However, if the size of input and output exceeds the capacity of internal memory, each access will take an individual I/O operation to disk, leading to a performance degradation. The DSA-IS algorithm solves this problem by performing the following two steps during a reduction/induction phase:

S1 Split x into blocks and sort substrings/suffixes of each block by calling SA-IS. The heading characters in need are copied to external memory in their access order.
S2 Sort the substrings/suffixes by their heading characters and the ranks of their successors in an external-memory heap.

Because all the substrings/suffixes in the heap are sorted already, S2 determines the rank of a substring/suffix when popping it from the heap. When inducing the predecessor of current top substring/suffix, the heading character in need is retrieved from external memory by a sequential I/O operation. As shown in Section 5, our program for DSA-IS requires less disk space than that for eSAIS, but the former runs slower than the latter due to the large I/O volume for sorting and naming S*-type substrings during the reduction

phase. To improve the performance, we propose new substring sorting and naming methods in the following.

3.2 Improvements on DSA-IS

All the S*-type substrings are classified into long and short categories with respect to whether or not containing more than D characters. The new substring sorting method mainly consists of the three steps below:

S1′ Sort the long in each block, copy the heading characters in need to external memory by S1. Meanwhile, sort the short in each block, copy the short to external memory in their sorted order.

S2′ Sort the long in x by S2, copy the leftmost D characters of the long to external memory in their sorted order.

S3′ Merge the short and the long by a multi-way sorter.

The sorted short S*-type substrings in each block are organized as a sequence in external memory after S1′, while the sorted long S*-type substrings in x are also organized as a sequence in external memory after S2′. Assume x is split into k blocks, the multi-way sorter in S3′ maintains an internal-memory heap to determine the lexical order of substrings belonging to different sequences. The heap contains at most $k+1$ substrings at any point of time. It compares any two substring in $\mathcal{O}(D)$ time by conducting a literal string comparison[1].

The above sorting method has a good performance if D is small and the majority of S*-type substrings in x are short. This is commonly satisfied in real-world datasets. We next describe a method for naming the S*-type substrings when they are being sorted according to S1′-S3′. The key point is to check equality of two suffixes successively popped from the heap. For this, we literally compare the two substrings in $\mathcal{O}(D)$ time if either of them is short; otherwise, we compare their names in $\mathcal{O}(1)$ time instead. Recall that the rank of each long S*-type substring is determined when inducing its predecessor, this value is reused to represent its name herein[2]. The experimental results in Section 5 demonstrate that, by using the new substring sorting and naming methods, the adapted DSA-IS, called DSA-IS+, only takes half as much disk space as eSAIS.

4 Checkers

4.1 Prior Art

We describe below the main idea of the existing checker presented in [5].

Lemma 4.1. $sa[0,n)$ is the SA for $x[0,n)$ if and only if the following conditions are satisfied:
(1) sa is a permutation of $[0,n)$.
(2) $\mathsf{r_i} < \mathsf{r_j} \Leftrightarrow (x[i], r_{i+1}) < (x[i], r_{j+1})$ for $i,j \in [0,n)$ and $i \neq j$, where r_i and r_j represent the ranks of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ among all the suffixes, respectively.

1. If the leftmost D characters of a long S*-type substring is equal to a short S*-type substring, then the short is lexicographically greater than the long.

2. A similar technique has been used to merge the substring sorting and naming process in SAIS-PQ.

Proof: If condition (1) is true, then sa is a permutation of all the suffixes in x. If condition (2) is true, then the lexical order of any two neighboring suffixes in sa is determined by comparing their heading characters and the ranks of their successors. As a result, all the suffixes in x are sorted according to the lexical order of their heading characters and successors.

□

The disk-based implementation of this checker conducts two passes of integer sorts and each sort arranges the order of $\mathcal{O}(n)$ fixed-size tuples in external memory. As can be observed from Section 5, the peak disk use and the I/O volume for an SA encoded by 40-bit integers are around $26n$ and $53n$, respectively.

4.2 Proposals

Recall that, an IS suffix sorting algorithm recursively executes the reduction phase to compute the reduced string x_1 until x_1 contains no duplicate characters. Afterward, it produces sa_1 from x_1 and recursively executes the induction phase to compute sa until reaching the top recursion level, where the induction phase consists of the following steps:

S1″ sort the starting positions of all the S*-type suffixes with their ranks indicated by sa_1 to produce sa^* .

S2″ Clear sa. Scan sa^* leftward with i decreasing from n_1-1 to 0. For each scanned item $sa^*[i]$, insert it into the rightmost empty position of $\mathsf{sa\_bkt_S}(x[sa^*[i]])$.

S3″ Scan sa rightward with i increasing from 0 to $n-1$. For each scanned non-empty item $sa[i]$, insert $sa[i]-1$ into the leftmost empty position of $\mathsf{sa\_bkt_L}(x[sa[i]-1])$ if $t[sa[i]-1]=0$.

S4″ Clear $\mathsf{sa\_bkt_S}(c)$ for $c \in \Sigma$. Scan sa leftward with i decreasing from $n-1$ to 0. For each scanned non-empty item $sa[i]$, insert $sa[i]-1$ into the rightmost empty position of $\mathsf{sa\_bkt_S}(x[sa[i]-1])$ if $t[sa[i]-1]=1$.

A running example of S2″-S4″ is shown in Fig. 1. Given sa^* is already known, line 6 inserts each S*-type suffix into the corresponding bucket. For example, $\mathsf{suf}(2)$, $\mathsf{suf}(5)$, $\mathsf{suf}(8)$ are sequentially placed into the current rightmost empty position of $\mathsf{sa\_bkt}(i)$, where the insertion order corresponds to their sorted order indicated by sa^*. The next step is to find the leftmost position of each bucket (marked by the symbol ∧) and scan sa rightward for inducing the order of L-type suffixes. For this, we first check $sa[0]=14$ (marked by the symbol @) and induce the predecessor of $\mathsf{suf}(14)$ in lines 8-9. Because $x[13]=i$ is L-type, we put $\mathsf{suf}(13)$ into the current leftmost empty position in $\mathsf{sa\_bkt_L}(i)$. To step through sa in this way, we get all the L-type suffixes sorted in line 17. Afterward, we find the rightmost position of each bucket and scan sa leftward for inducing the order of S-type suffixes. When scanning $sa[14]=4$ in lines 19-20, we see $x[3]=i$ is S-type and thus put $\mathsf{suf}(3)$ into the current rightmost empty position in $\mathsf{sa\_bkt_S}(i)$. Following the same idea, we get all the S-type suffixes sorted in line 28. Following the discussion, we show in Lemma 4.2 a set of sufficient and necessary conditions for checking SA.

Lemma 4.2. For any IS suffix sorting algorithm, its output $sa[0,n)$ is the SA for $x[0,n)$ if and only if the following conditions are satisfied at the top recursion level:
(1) sa^* is correctly computed.

00	p:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
01	x:	m	m	i	i	s	i	i	s	i	i	p	p	i	i	#
02	t:	0	0	1	1	0	1	1	0	1	1	0	0	0	0	1
03	sa^*:	14	8	5	2											
04	insert the sorted S*-type suffixes:															
05	bkt:	#						i					m		p			s
06	sa:	14	-1	-1	-1	-1	-1	8	5	2	-1	-1	-1	-1	-1	-1
07	induce L-type suffixes:															
08	sa:	14	-1	-1	-1	-1	-1	8	5	2	-1	-1	-1	-1	-1	-1
09		@^	^								^		^		^	
10		14	13	-1	-1	-1	-1	8	5	2	-1	-1	-1	-1	-1	-1
11		^	@	^							^		^		^	
12		14	13	12	-1	-1	-1	8	5	2	-1	-1	-1	-1	-1	-1
13		^		@	^						^		^		^	
14		14	13	12	-1	-1	-1	8	5	2	-1	-1	11	-1	-1	-1
15		^			^			@			^			^	^	
16									...	...						
17		14	13	12	-1	-1	-1	8	5	2	1	0	11	10	7	4
18	induce S-type suffixes:															
19	sa:	-1	13	12	-1	-1	-1	-1	-1	-1	1	0	11	10	7	4
20		^								^		^		^		@^
21		-1	13	12	-1	-1	-1	-1	-1	3	1	0	11	10	7	4
22		^							^			^		^	@	^
23		-1	13	12	-1	-1	-1	-1	6	3	1	0	11	10	7	4
24		^						^				^		@^		^
25		-1	13	12	-1	-1	-1	9	6	3	1	0	11	10	7	4
26		^						^				^	@	^		^
27									...	...						
28		14	13	12	8	5	2	9	6	3	1	0	11	10	7	4
29	$\overline{sa^*}$:	14			8	5	2									

Fig. 1. An example for inducing sa from sa^*.

(2) S1″-S4″ are correctly implemented.

Proof: Given two suffixes placed at $sa[i]$ and $sa[j]$ and their successors placed at $sa[p]$ and $sa[q]$ at the top recursion level, we prove the statement as follows.

Because S1″-S4″ are correctly implemented, $i < j \iff (x[sa[i]], p) < (x[sa[j]], q)$. This satisfies the second condition of Lemma 4.1.

Further, suppose $sa[i] = sa[j]$ and $i \neq j$, then $sa[p] = sa[q]$ and $p \neq q$. Repeating this reasoning process by replacing (i, j) with (p, q) until $\mathsf{suf}(sa[i])$ is S*-type, then sa^* must contain duplicate elements. However, each element in sa^* is unique as sa^* is corrected computed. This leads to a contradiction.

□

Lemma 4.2 shows that we can check the correctness of sa^* instead of sa at the top recursion level to perform SA verification if the first condition of Lemma 4.2 is always true. In fact, the code snippet for the induction phase at the top recursion level only consists of tens of C++ code lines. This is considerably smaller than the whole program for a disk-based suffix sorting algorithm. Following the idea, we assume S1″-S4″ are correctly implemented and propose two methods for checking computation errors caused by implementation bugs and other malfunctions. The algorithms designed by these methods can be seamlessly integrated into any IS suffix sorting algorithm for building and checking an SA at the same time.

4.2.1 Method A

Method A is based on Lemma 4.3, which is an extension of Lemma 4.2.

Lemma 4.3. Assume the induction phase at the top recursion level is correctly implemented, the output $sa[0, n)$ of an IS suffix sorter is the SA for the input string $x[0, n)$ if the following conditions are satisfied:
(1) sa^* is correct.
(2) $sa[i]$ is equal to the value calculated by S1″-S4″ for $i \in [0, n)$.

The first condition of Lemma 4.3 can be checked by a sparse SA checker like [11]. The second condition intends for detecting malfunctions other than implementation bugs (e.g., I/O errors). This is checked by determining whether the induced and scanned values for each suffix are equal[3]. The problem to be solved here is that when a suffix is induced into a bucket, its corresponding value in sa may not be scanned at once. Our solution is to ensure the sequence of values placed at a bucket is identical to that scanned latter. For the purpose, we use a fingerprinting function to increasingly compute the fingerprints of both sequences and check their equality in constant time at the end of the induction phase. If the two fingerprints for each bucket are equal, then the second condition of Lemma 4.3 will be seen with a high probability. As a result, sa can be built and probabilistically checked at the same time.

It was reported in [2], [12] that the IS method can be used to construct LCP array as well. For any two neighboring suffixes in sa, their LCP-value is zero if they start with different characters; otherwise, the LCP-value of them is one greater than that of their successors. This implies the possibility for extending Method A to check both suffix and LCP arrays. We design Algorithm 2 to check sa and lcp when they are induced from sa^* and lcp^* at the top recursion level of an IS builder. In this algorithm, $\mathsf{sa_{l1}}(c)$ and $\mathsf{sa_{l2}}(c)$ are two sequences respectively induced into $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{sa\_bkt_S}(c)$, while $\mathsf{sa_{S1}}(c)$ and $\mathsf{sa_{S2}}(c)$ are two sequences respectively scanned from $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{sa\_bkt_S}(c)$. These values are used to check the second condition of Lemma 4.3 later. We also compute $\mathsf{lcp_{l1}}(c)/\mathsf{lcp_{l2}}(c)$ and $\mathsf{lcp_{S1}}(c)/\mathsf{lcp_{S2}}(c)$ and check their equality in the meantime.

4.2.2 Method B

According to Lemma 4.4, Method B uses a different way to check the correctness of sa^*. Before our presentation, we first introduce a notation called $\overline{sa^*}$. The same as sa^*, $\overline{sa^*}$ also represents the suffix array for all the S*-type suffixes. Both sa^* and $\overline{sa^*}$ are computed during the induction phase at the top recursion level, where the former is calculated from sa_1 in S1″ and the latter is calculated when inducing the S*-type suffixes into sa in S4″.

Lemma 4.4. Assume the induction phase at the top recursion level is correctly implemented, the output $sa[0, n)$ of an

3. Notice that each suffix induced into sa will be latter scanned for inducing the order of its predecessor during S3″ and S4″.

Algorithm 2: The Algorithm Based on Lemma 4.3.

1 **Function** CheckByMethodA(x, sa^*, lcp^*)
2 Verify sa^* and lcp^* by the checker presented in [11].
3 Compute the fingerprints of $\mathsf{sa_{l1}}(c)$, $\mathsf{sa_{S1}}(c)$, $\mathsf{lcp_{l1}}(c)$ and $\mathsf{lcp_{S1}}(c)$ when inducing the order and the LCP-values of L-type suffixes.
4 Check if $\mathsf{sa_{l1}}(c) = \mathsf{sa_{S1}}(c)$ and $\mathsf{lcp_{l1}}(c) = \mathsf{lcp_{S1}}(c)$.
5 Compute the fingerprints of $\mathsf{sa_{l2}}(c)$, $\mathsf{sa_{S2}}(c)$, $\mathsf{lcp_{l2}}(c)$ and $\mathsf{lcp_{S2}}(c)$ when inducing the order and the LCP-values of S-type suffixes.
6 Check if $\mathsf{sa_{l2}}(c) = \mathsf{sa_{S2}}(c)$, $\mathsf{lcp_{l2}}(c) = \mathsf{lcp_{S2}}(c)$.

IS suffix sorter is the SA for the input string $x[0, n)$ if the following conditions are satisfied:
(1) $sa^*[i]$ is a permutation of all the S*-type suffixes.
(2) $sa^*[i] = \overline{sa^*}[i]$ for $i \in [0, n_1)$.
(3) $sa[i]$ is equal to the value calculated by S1″-S4″ for $i \in [0, n)$.

Proof:
We prove the sufficiency according to the following two cases.

Case 1: Suppose all the S*-type suffixes are correctly sorted in sa^*, then sa is right based on Lemma 4.3.

Case 2: Suppose any two S*-type suffixes are not correctly sorted, i.e., $\mathsf{suf}(sa^*[i_0]) > \mathsf{suf}(sa^*[j_0])$ and $i_0 < j_0$. By condition (2), we have $\mathsf{suf}(\overline{sa^*}[i_0] > \mathsf{suf}(\overline{sa^*}[j_0])$. Given that the order of $\mathsf{suf}(\overline{sa^*}[i_0])$ and $\mathsf{suf}(\overline{sa^*}[j_0])$ are induced from $\mathsf{suf}(sa^*[i_1])$ and $\mathsf{suf}(sa^*[j_1])$, then $\mathsf{sub}(\overline{sa^*}[i_0], sa^*[i_1])$ and $\mathsf{sub}(\overline{sa^*}[j_0], sa^*[j_1])$ are two S*-type substrings and there must be $\mathsf{suf}(sa^*[i_1]) > \mathsf{suf}(sa^*[j_1])$ and $i_1 < j_1$. Repeating this reasoning process, because condition (1), we must see $\mathsf{suf}(sa^*[i_k]) > \mathsf{suf}(sa^*[j_k])$ and $\mathsf{suf}(sa^*[i_{k+1}]) < \mathsf{suf}(sa^*[j_{k+1}])$, where $i_k < j_k$ and $i_{k+1} < j_{k+1}$. However, given $\mathsf{suf}(sa^*[i_{k+1}]) < \mathsf{suf}(sa^*[j_{k+1}])$, the inducing process will produce $\mathsf{suf}(\overline{sa^*}[i_k]) < \mathsf{suf}(\overline{sa^*}[j_k])$, which implies $\mathsf{suf}(sa^*[i_k]) < \mathsf{suf}(sa^*[j_k])$ because condition (2), leading to a contradiction.

□

The first condition of Lemma 4.4 is naturally satisfied when computing sa^* from sa_1 in S1″. The second condition is probabilistically checked by computing and comparing the fingerprints of sa^* and $\overline{sa^*}$. The fingerprint of sa^* is computed when placing the elements of sa^* into sa in S2″, while the fingerprint of $\overline{sa^*}$ is computed when inducing the S*-type suffixes into sa in S4″. We design Algorithm 3 to check if the final SA observes Lemma 4.4.

4.3 Fingerprinting Function

The proposed two methods check equality of two arrays (or sub-arrays) by comparing their fingerprints, which can be calculated using a hash function. Notice that two equal arrays must be mapped to an identical hash value, but the inverse is not always true. To lower the error probability of a false match, we prefer using the Karp-Rabin fingerprinting function to compute these hash values, called fingerprints, following Formulas 4.5-4.6, where L is a prime and δ is an integer randomly chosen from $[1, L)$. By setting L to a large value, the error probability can be reduced to a negligible level. In Fig. 2, we depict an example for computing the integer array A identical to sa^* and $\overline{sa^*}$ in Fig. 1. Given $L = 23$ and $\delta = 5$, we first compute the fingerprint of $A[0, 0]$ in line 4. Because $\mathsf{fp}(\mathsf{A}[0, -1]) = 0$, $\mathsf{fp}(\mathsf{A}[0, 0]) = (0 \cdot 5 + 14) \bmod 23 = 14$. By iteratively computing the fingerprints of the prefixes of A, we finally obtain the fingerprint of A in line 7.

Formula 4.5. $\mathsf{fp}(A[0, -1]) = 0$.

Formula 4.6. $\mathsf{fp}(A[0, i]) = \mathsf{fp}(A[0, i-1]) \cdot \delta + A[i] \mod L$ for $i \geq 0$.

01	p:	0	1	2	3
02	A:	14	8	5	2

03 compute $\mathsf{fp}(A[0, p])$ with $L = 23$, $\delta = 5$:
04 $\mathsf{fp}(A[0, 0]) = \mathsf{fp}(A[0, -1]) * 5 + A[0] \mod 23 = 14$
05 $\mathsf{fp}(A[0, 1]) = \mathsf{fp}(A[0, 0]) * 5 + A[1] \mod 23 = 9$
06 $\mathsf{fp}(A[0, 2]) = \mathsf{fp}(A[0, 1]) * 5 + A[2] \mod 23 = 4$
07 $\mathsf{fp}(A[0, 3]) = \mathsf{fp}(A[0, 2]) * 5 + A[3] \mod 23 = 22$

Fig. 2. An example for calculating fingerprints by Karp-Rabin fingerprinting function.

4.4 Complexity Analysis

Assume the induction phase at the top recursion level is correctly implemented, Algorithm 2 employs the method proposed in [11] to ensure the correctness of sa^* and lcp^*. To be specific, for two neighboring suffixes $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ in sa^*, their LCP-value ℓ in lcp^* is right if $\mathsf{sub}(i, i + \ell - 1) = \mathsf{sub}(j, j + \ell - 1)$ and $x[i + \ell] \neq x[j + \ell]$, where the equality of $\mathsf{sub}(i, i+\ell-1)$ and $\mathsf{sub}(j, j+\ell-1)$ is checked by comparing the fingerprints of these substrings. Given that ℓ is correct, the lexical order of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ is checked by comparing $x[i + \ell]$ with $x[j + \ell]$. This method computes the fingerprints of all the target substrings within a sorting complexity proportional to n_1. Because at most one out of every two suffixes is S*-type, the checking overhead is much less than building consumption. By contrast, Algorithm 3 takes overall $\mathcal{O}(n)$ to compute all the required fingerprints.

5 Experiments

For performance comparison, we engineer DSA-IS and DSA-IS+ by the STXXL's containers (vector, sorter, priority queue and stream). The experimental platform is a desktop

Algorithm 3: The Algorithm Based on Lemma 4.4.

Function CheckByMethodB(x, sa^*)
  Compute the fingerprints of sa^*, $\mathsf{sa_{l1}}(c)$ and $\mathsf{sa_{S1}}(c)$ when inducing the order of L-type suffixes.
  Check if $\mathsf{sa_{l1}}(c) = \mathsf{sa_{S1}}(c)$ by comparing their fingerprints.
  Compute the fingerprints of $\overline{sa^*}$, $\mathsf{sa_{l2}}(c)$ and $\mathsf{sa_{S2}}(c)$ when inducing the order of S-type suffixes.
  Check if $\mathsf{sa_{l2}}(c) = \mathsf{sa_{S2}}(c)$ by comparing their fingerprints.
  Check if $sa^* = \overline{sa^*}$ by comparing their fingerprints.

computer equipped with an Intel Xeon E3-1220 V2 CPU, 4GiB RAM and 500GiB HD. All the programs are complied by gcc/g++ 4.8.4 with -O3 options under Ubuntu 14.04 64-bit operating system. In our experiments, three performance metrics are investigated for the programs running on the corpora listed in Table 1, where each metric is measured as a mean of two runs.

- construction time (CT): the running time, in units of microseconds per character.
- peak disk use (PDU): the maximum disk space requirement, in units of bytes per character.
- I/O volume (IOV): as the term suggests, in units of bytes per character.

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

Corpora	n	$\|\|\Sigma\|\|$	Description
guten	22.5	256	Gutenberg, at http://algo2.iti.kit.edu/bingmann/esais-corpus.
enwiki	74.7	256	Enwiki, at https://dumps.wikimedia.org/enwiki, dated as 16/05/01.
proteins	1.1	27	Swissprot database, at http://pizzachili.dcc.uchile.cl/texts/protein, dated as 06/12/15.
uniprot	2.5	96	UniProt Knowledgebase release 4.0, at ftp://ftp.expasy.org/databases/.../complete, dated as 16/05/11.

5.1 Building Performance

Because fSAIS is not available online, we use eSAIS as a baseline for analyzing the performance of DSA-IS and DSA-IS+, where the program for eSAIS is also implemented by the STXXL library. Fig. 3 shows a comparison between the programs for these three algorithms in terms of the investigated metrics. As depicted, the program for DSA-IS requires less disk space than that for eSAIS when running on "enwiki" and "guten". In details, the peak disk use of DSA-IS and eSAIS are around $18n$ and $24n$, respectively. However, eSAIS runs much faster than DSA-IS due to the different I/O volumes. In order for a deep insight, we collect in Table 2 the statistics of their I/O volumes in the reduction and induction phases. As can be seen, although DSA-IS and eSAIS have similar performances when sorting suffixes in the induction phase, the latter consumes much less I/O volume than the former when sorting substrings in the reduction phase. More specifically, the mean ratio of induction I/O volume to reduction I/O volume are 0.23 and 0.71 for them, respectively. We can also see from the same figure that DSA-IS+ achieves a substantial improvement against DSA-IS, it runs as fast as eSAIS and takes half as much disk space as the latter. This is because the reduction I/O volume for DSA-IS+ is only half as much as that for DSA-IS (Table 2). Notice that the new substring sorting and naming methods adopted by DSA-IS+ take effect when most of the S*-type substrings are short. From our experiments, given $D = 8$, the ratio of long S*-type substrings in the investigated corpus nearly approaches one hundred percent, indicating that these methods are practical for real-world datasets.

5.2 Checking Performance

For evaluation, we integrate Method B into DSA-IS+ to constitute "Solution A" and compare it with "Solution B" composed of eSAIS and the existing checking method in [5]. Fig. 4 gives a glimpse of the performance of two solutions on various corpora. It can be observed that, the time, space and I/O volume for verification by Method B is negligible in comparison with that for construction by DSA-IS+, while the overhead for checking SA in Solution B is relatively large. Table 3 shows the performance breakdown of Solution B, where the checking time is one-fifth as the running time of the plain eSAIS and the peak disk use for verification is also a bit larger than that for construction. As a result, the combination of DSA-IS+ and Method B can build and check an SA in better total time and space.

5.3 Discussion

Rather than designing an I/O layer for efficient I/O operations, we currently use the containers provided by the STXXL library to perform reading, writing and sorting in external memory, these containers do not free the disk space for storing temporary data even if it is not needed any more, leading to a space consumption higher than our expectation. This is an implementation issue that can be solved by storing the data into multiple files and deleting each file when it is obsolete. In this way, our program can be further improved to achieve a space performance comparable to fSAIS. Our next paper will describe a novel disk-based IS suffix sorter that only takes 1n work space excluding the disk space for storing input and output.

6 Conclusion

By assuming the induction phase at the top recursion level is correctly implemented, we proposed two methods that enable any IS suffix sorter to build and check an SA simultaneously. The probabilistic algorithm designed by the second method is rather lightweight, it takes negligible time and space compared with the existing IS suffix sorting and

TABLE 2
A Comparison of Reduction and Induction I/O Volumes Amongst DSA-IS, DSA-IS+ and eSAIS on enwiki

	eSAIS				DSA-IS				DSA-IS+ ($D = 4$)			
Size	Red.	Ind.	Total	Ratio	Red.	Ind.	Total	Ratio	Red.	Ind.	Total	Ratio
1G	36.6	132.8	169.4	0.27	81.3	109.6	190.9	0.74	45.4	91.7	137.1	0.33
2G	36.0	141.9	177.9	0.25	83.5	111.6	195.1	0.75	47.2	93.4	140.6	0.34
4G	35.6	152.1	187.7	0.23	94.3	144.1	238.4	0.65	54.1	111.5	165.6	0.33
8G	35.2	165.7	200.9	0.21	107.8	159.6	267.4	0.68	60.1	122.1	182.2	0.33
16G	35.0	172.1	207.1	0.20	121.9	166.1	288.0	0.73	62.7	128.7	191.4	0.33

TABLE 3
Performance Breakdown of Solution B on various Corpora

Corpus	checking			building		
	PDU	IOV	CT	PDU	IOV	CT
enwiki_16G	26.0	53.0	0.71	23.5	205.6	3.49
guten_16G	26.0	53.0	0.79	23.4	195.2	3.20
uniprot	25.9	53.0	0.74	22.7	162.0	2.50
proteins	25.9	53.0	0.58	24.1	172.3	2.33

checking algorithms. We also made our first attempt to improve the space performance of DSA-IS using new substring and naming methods. Our experimental results show that the program for the adapted algorithm DSA-IS+ runs as fast as that for eSAIS and consumes only half as much disk space as the latter on various real-world datasets. We are now designing and implementing a novel IS suffix sorter that takes no more than 1n work space on external memory model. Theoretically, this suffix sorter has a better space performance than fSAIS under the same circumstances.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] W. J. Liu, G. Nong, W. H. Chan, and Y. Wu, "Induced Sorting Suffixes in External Memory with Better Design and Less Space," in *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval*, London, UK, September 2015, pp. 83–94.

[5] 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.

[6] P. Ferragina, T. Gagie, and G. Manzini, "Lightweight Data Indexing and Compression in External Memory," *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.

[7] 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.

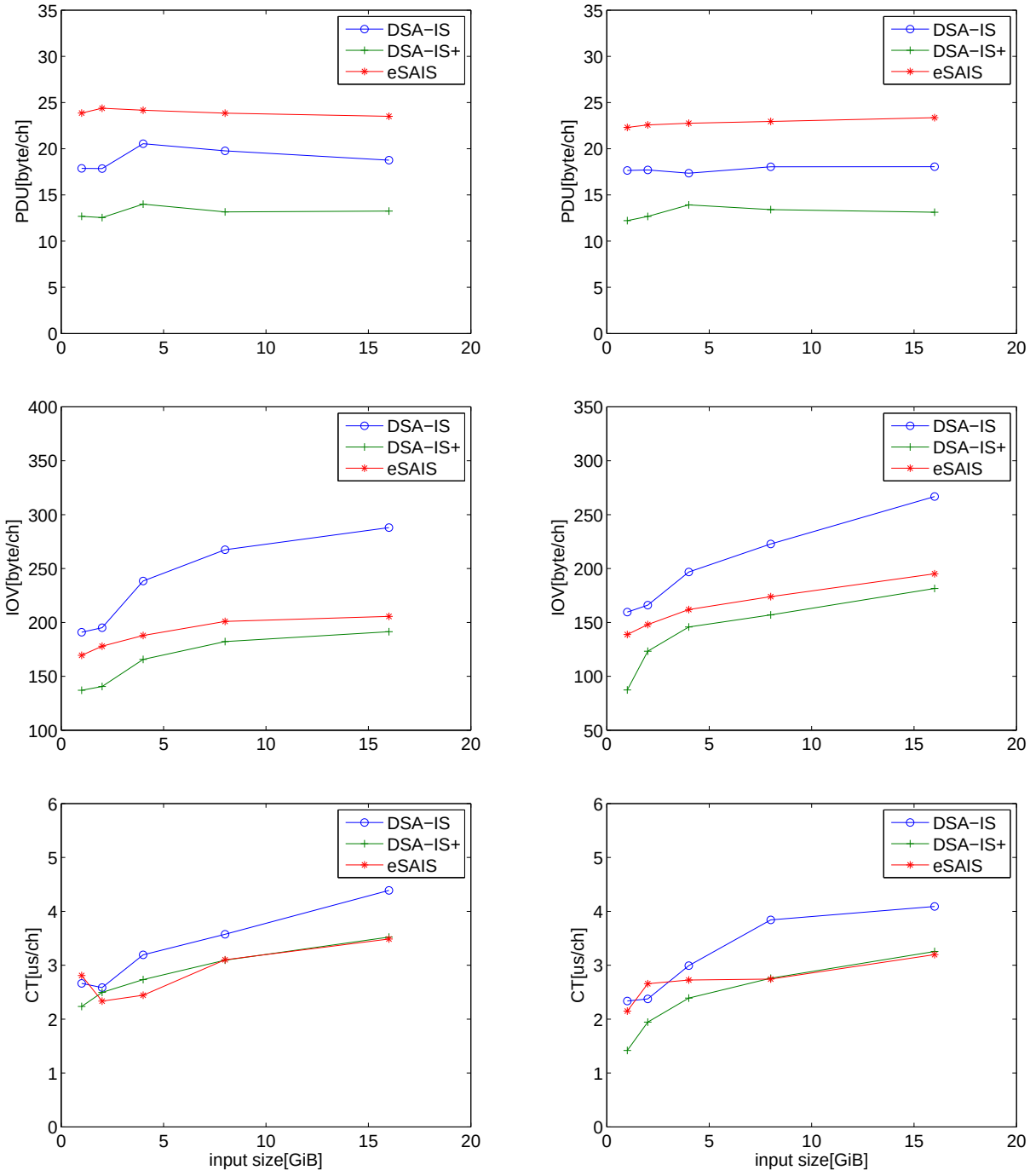
[8] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel External Memory Suffix Sorting," in *In proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching*, Ischia Island, Italy, July 2015, pp. 329–342.

[9] 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.

[10] 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.

[11] Y. Wu, G. Nong, W. H. Chan, and L. B. Han, "Checking Big Suffix and LCP Arrays by Probabilistic Methods," *IEEE Transactions on Computers*, 2017.

[12] J. Fischer, "Inducing the LCP-array," in *In Workshop on Algorithms and Data Structures*, 2011, pp. 374–385.



(a) enwiki (b) guten

Fig. 3. A comparison of DSA-IS, DSA-IS+ and eSAIS on guten and enwiki in terms of peak disk usage, I/O volume and construction time, where $D = 4$ and the input size varies in $\{1, 2, 4, 8, 16\}$ GiB.

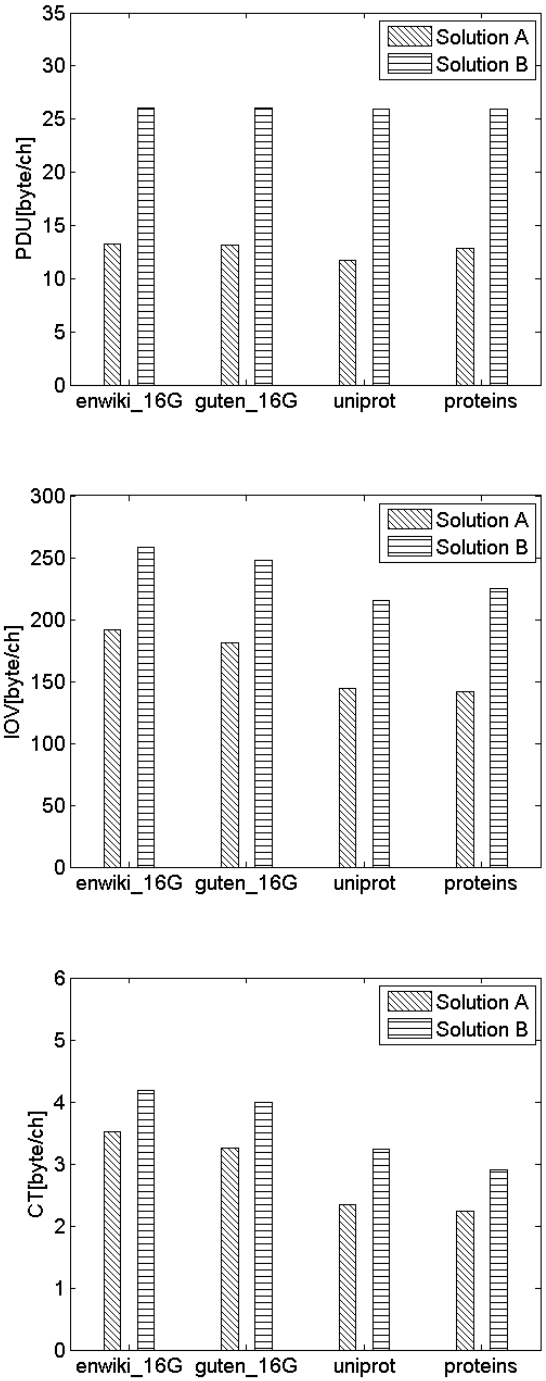


Fig. 4. A comparison of Solutions A and B on various corpora in terms of peak disk usage, I/O volume and construction time, where $D = 4$.