Building and Checking Suffix Array Using Induced Sorting Method

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

Abstract—Suffix array (SA) can be built by the induced sorting (IS) method on both internal and external memory models. We propose two methods that enable any IS suffix sorter to build and check an SA simultaneously. The first method can check both suffix and longest common prefix (LCP) arrays for strings drawn from constant alphabets, while the second method is applicable to checking SA for strings of any alphabets. By combining our methods with the Karp-Rabin fingerprinting function, we design two probablistic algorithms that perform verification with a neligible error probability. These algorithms are rather lightweight in terms of time and space. Particularly, the algorithm designed by the second method only takes linear time and constant space when running in external memory. For a further study, we integrate the checking algorithm designed by the second method into the existing suffix sorter DSA-IS to evaluate the checking overhead, where the time, space and I/O consumptions for verification are considerably smaller than that for construction in the experiments. This convinces us that the proposed checking methods, together with the state-of-the-art IS suffix sorters, could constitute efficient solutions for both construction and verification.

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

1 Introduction

A suffix array can be built in linear time and space by the internal-memory algorithm SA-IS [1]. According to the IS principle, the relative order of two unsorted suffixes is determined by sequentially comparing the heading characters and the sorted succeeding suffixes starting with their second characters. Recently, the IS method has been applied to designing three disk-based suffix sorting algorithms eSAIS [2], DSA-IS [3] and SAIS-PQ [4], which have a better time complexity than the other alternatives (e.g., DC3 [5], bwt-disk [6], SAscan [7] and pSAscan [8]) but suffer from a bottleneck due to the large disk space for obtaining the heading characters of unsorted suffixes and the ranks of their successors in a disk-friendly way. More specifically, the average peak disk use to construct an SA encoded by 40-bit integers for pSAscan is $7.5n$, while that for eSAIS, DSA-IS and SAIS-PQ are $24n$, $20n$ and $15n$, respectively. The poor space performance is mainly because that the current programs for these algorithms fail to free the disk space for temporary data even when the data is no longer to use. A dramatic improvement can be achieved by splitting a file into multiple pieces and deleting each piece immediately when the temporary data in it is not needed any more. This technique has been used to implement a new IS suffix sorting algorithm fSAIS [9]. As reported, the engineering of fSAIS consumes no more than $8n$ disk space for constructing an SA encoded by 40-bit integers, indicating a great potential for optimizing our programs for DSA-IS and SAIS-PQ.

A constructed SA should be checked to avoid computation errors caused by implementation bugs and/or hardware malfunctions. Currently, the software packages for DC3 and eSAIS provide users a checker designed by the method proposed in [5]. The overhead for this checker is rather high because it takes two external-memory sorts for arranging $\mathcal{O}(n)$ fixed-size tuples according to their integer keys. Against the background, we propose in this paper two methods that enable any IS suffix sorter to check an SA when it is being built. The first method can check both suffix and LCP arrays for strings drawn from constant alphabets, while the second method can check SA for strings drawn from any alphabets. We augment these methods with the Karp-Rabin fingerprinting function [10] to design two probabilistic algorithms, in terms of that their checking results are wrong with a negligible probability. These algorithms are rather lightweight in terms of time and space. Particularly, the algorithm designed by the second method only takes linear time and constant space to run using external memory. For a further study, we integrate this algorithm into DSA-IS to evaluate its checking overhead, where the design of DSA-IS has been adapted by new substring sorting and naming methods for a better performance. As demonstrated in our experiments, the time, space and I/O consumptions for verification by this algorithm are negligible to that for construction by the adapted DSA-IS. We are convinced that the proposed SA checkers, together with the state-of-the-art IS suffix sorters, could constitute efficient solutions for both construction and verification.

The rest of this paper is organized as follows. Section 2 introduces some basic notations and symbols. Section 3 gives an overview of IS suffix sorting algorithms and the details of new substring sorting and naming methods. Section 4 describes the ideas of checking methods and algo-

- *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.*

rithms. 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. A substring or suffix is of the same type as its heading character. In this paper, we only consider substrings ending with an S*-type 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 $x[i]$. Accordingly, we define the predecessor-successor relationship between $\mathsf{suf}(i)/\mathsf{sub}(i,j)$ and $\mathsf{suf}(i+1)/\mathsf{sub}(i+1,j)$.

We partition x into S*-type substrings and replace each substring with its name to produce the reduced string x_1, where the name indicates the rank of the corresponding S*-type substring among all.

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

The LCP array lcp indicates the LCP-values of all the neighboring suffixes in sa, where $lcp[0] = 0$ and $lcp[i] = \ell$ for $i \in [1,n)$, where ℓ is the length of longest common prefix of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[i-1])$.

All the suffixes in sa are naturally partitioned into multiple buckets. Each bucket occupies a contiguous interval in sa and contains all the suffixes starting with a certain character. Without loss of generality, we denote by $\mathsf{sa\_bkt}(c_0)$ the bucket for suffixes starting with c_0. Notice that $\mathsf{sa\_bkt}(c_0)$ can be further divided into two parts, where the left part $\mathsf{sa\_bkt_L}(c_0)$ and the right part $\mathsf{sa\_bkt_S}(c_0)$ contain the L-type and S-type suffixes, respectively. We also partition lcp into buckets such that $\mathsf{lcp\_bkt}(c_0)$, $\mathsf{lcp\_bkt_L}(c_0)$ and $\mathsf{lcp\_bkt_S}(c_0)$ record the LCP-values of suffixes in $\mathsf{sa\_bkt}(c_0)$, $\mathsf{sa\_bkt_L}(c_0)$ and $\mathsf{sa\_bkt_S}(c_0)$, respectively.

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 overview of IS suffix sorting algorithms .

Input: x
Output: sa

```
/* Reduction Phase */
Sort S*-type substrings by the IS method.
Name the sorted S*-type substrings to produce x_1.

/* Check Recursion Condition */
if exist two equal characters in x_1 then
    Recursively call the reduction phase on x_1.
end

/* Induction Phase */
Sort suffixes by the IS method.
```

3.1 Introduction to IS Suffix Sorting Algorithms

As shown in Algorithm 1, an IS suffix sorting algorithm first performs a reduction phase to produce the reduced string x_1. If there exist two equal characters in x_1, then it recursively calls the reduction phase with x_1 as input; otherwise, all the S*-type suffixes in x are already sorted and it performs an induction phase to produce the final sa. During the execution of two phases, the order of substrings/suffixes is determined by comparing the heading characters and the ranks of their successors according to the IS principle. This involves frequent random accesses to x and sa for retrieving the heading characters of unsorted substrings/suffixes and the ranks of their sorted successors. It can be done very quickly if both x and sa are wholly loaded into RAM; otherwise, each disk access takes an individual I/O operation, leading to a severe performance degradation. This problem is solved in DSA-IS by conducting two steps:

S1 Split x into blocks and sort substrings/suffixes of each block by calling SA-IS, the heading characters in use are copied to external memory in their access order.

S2 Sort the substrings/suffixes of x by their heading characters and the ranks of their successors. Organize the sorted substrings/suffixes in an external-memory heap and induce their predecessors into the heap according to the IS principle, where the heading characters of the induced substrings/suffixes are retrieved from external memory by sequential I/O operations.

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 caused by the large I/O volume for sorting and naming S*-type substrings during the reduction phase. For this, we present new substring sorting and naming methods in the next subsection.

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 of each block by S1. During the process, copy the short to external memory in their sorted order.

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

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

After the first two steps, the short S*-type substrings of each block and the long S*-type substrings of x are sorted and separately organized as a sequence in external memory. Assume x is split into k blocks, the multi-way sorter in S3′ maintains an internal-memory heap to cache the current smallest of each sequence and continually retrieve the top item from the heap to determine the lexical order of substrings from different sequences. The heap contains at most $k+1$ substrings at any point of time and it compares any two substrings in $\mathcal{O}(D)$ time by a literal string comparison[1]. Theoretically, the sorting method has a good performance if the majority of S*-type substrings in x are short with a small D. This is commonly satisfied in practice, because the average length of S*-type substrings is typically small in real-world datasets.

In what follows, we describe a method for naming the S*-type substrings during the above sorting process. The key point here is to check equality of two substrings successively popped from the heap in S3′ as fast as possible. If either of the two substrings is short, our solution is to literally compare them in $\mathcal{O}(D)$ time. Otherwise, if both of them are long, then we assign names to all the long S*-type substrings using the technique proposed for SAIS-PQ in S2′ and compare two successively popped long S*-type substrings by their names. This technique is based on the idea that two substrings are of the same name if their heading characters and the names of their successors are both equal[2].

The experimental results in Section 5 indicate 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. Also, the peak disk use of the program for DSA-IS+ can be further reduced to a level comparable to that of the program for fSAIS.

4 Checkers

4.1 Prior Art

The software packages for DC3 and eSAIS provide users a checker to ensure the correctness of their outputs. Below we describe the main idea behind this checker and refer interested readers to [5] for more details.

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.

Proof: The first condition indicates that sa is a permutation of all the suffixes in x. The second condition indicates that the order of suffixes in sa corresponds to that of their heading characters and successors. Because a comparison between any two suffixes can be done by comparing their heading characters and successors, the above conditions are sufficient and necessary for SA verification. □

The disk-based implementation of this checker conducts two passes of integer sorts and each sort arranges the order of n fixed-size tuples using 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 perform the reduction phase to compute the reduced string s_1 until s_1 contains no duplicate characters. Afterward, it produces sa_1 from s_1 and recursively perform the induction phase to compute sa until reaching the top recursion level, where the induction phase executes mainly 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-Sp4 is shown in Fig. 1. Given sa^* is already known, line 6 inserts each S*-type suffix into the right part of the corresponding bucket and arranges them according to their sorted order indicated by sa^*. For example, $\mathsf{suf}(8)$, $\mathsf{suf}(5)$, $\mathsf{suf}(2)$ are placed into the right part of $\mathsf{sa\_bkt(i)}$ and arranged in their sorted order indicated by sa^*. Then, we find the leftmost position of each bucket (marked by the symbol ∧) and scan sa rightward for inducing the order of L-type suffixes. To do 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. Based on the discussion, we show in Lemma 4.2 a set of sufficient and necessary conditions for SA verification.

Lemma 4.2. $sa[0,n)$ is the SA for $x[0,n)$ if and only if the following conditions are satisfied:
(1) sa^* is correctly computed.

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. We refer interested readers to [4] for more information.

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 $sa[q]$, 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 $p \neq q$. Repeatedly replacing (i, j) with (p, q) until $\mathsf{suf(sa[i])}$ is S*-type, then sa^* contain duplicate elements. Because sa^* is correctly computed, each element in sa^* must be unique. This leads to a contradiction. □

The above lemma indicates that, if the first condition of Lemma 4.2 is true, we can check the correctness of sa^* instead of sa at the top recursion level to perform SA verification. It should be noticed that the code snippet for the induction phase at the top recursion level only consists of tens of C++ code lines. This is considerable 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 I/O errors caused by hardware malfunctions. The algorithms designed by these methods can be seamlessly integrated into any IS suffix sorting algorithms for building and checking an SA at the same time.

4.2.1 Method A

Our first method is directly extended from 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 sorting algorithm is the SA for the input string $x[0, n)$ if and only if the following conditions are satisfied:
(1) sa^* is correct.
(2) $sa[i]$ is equal to the value calculated by S1″-S4″.

The first condition of Lemma 4.3 can be checked by a sparse SA checker like [11]. Notice that each suffix induced into sa will be latter scanned for inducing the order of its predecessor during S2″ and S4″. The second condition intends for detecting I/O errors and can be checked by determining whether the induced and scanned values for each suffix are equal. 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 integrate the checking process into the building process, by ensuring the sequence of values placed at a bucket is identical to that scanned later. 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. In this way, we design Algorithm 1 to probabilistically check $sa[0, n)$ during the induction phase at the top recursion level of an IS suffix sorting algorithm.

Theorem 4.4. Assume the induction phase at the top recursion level is correctly implemented, the output $sa[0, n)$ for the input string $x[0, n)$ if and only if the following conditions are satisfied for all $c \in \Sigma$:
(1) sa^* is correct.
(2) The fingerprints of $\mathsf{sa_{l1}}(c)$ and $\mathsf{sa_{S1}}(c)$ are equal.
(3) The fingerprints of $\mathsf{sa_{l2}}(c)$ and $\mathsf{sa_{S2}}(c)$ are equal.

In Algorithm 1, $\mathsf{sa_{l1}}(c)$ and $\mathsf{sa_{l2}}(c)$ are two sequences respectively induced into $\mathsf{sa\_bkt_L}$ and $\mathsf{sa\_bkt_S}$, while $\mathsf{sa_{S1}}(c)$ and $\mathsf{sa_{S2}}(c)$ are two sequences respectively scanned from $\mathsf{sa\_bkt_L}$ and $\mathsf{sa\_bkt_S}$. It should be noticed that Method A can be also applied to checking lcp when it is being built from lcp^* following the IS principle [12].

4.2.2 Method B

As shown in Lemma 4.5, our second method uses a different way to check the correctness of sa^*. Before the 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 two integer arrays are computed at the top recursion level of an IS suffix sorting algorithm. To be specific, sa^* is computed from sa_1 and $\overline{sa^*}$ is retrieved from the constructed sa.

Lemma 4.5. Assume the induction phase at the top recursion level is correctly implemented, the output $sa[0, n)$ of an IS suffix sorting algorithm is the SA for the input

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

Proof: We only prove the sufficiency as the necessity is clear.

Case 1: Suppose all the S*-type suffixes are correctly sorted, sa is correct by Lemma 4.3.

Case 2: Notice that the suffix inducing process at the top level is correctly implemented by Assumption xxx. 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 $\mathsf{suf}(\overline{sa^*}[i_0]) > \mathsf{suf}(\overline{sa^*}[j_0])$. Because the order of $\mathsf{suf}(\overline{sa^*}[i_0])$ and $\mathsf{suf}(\overline{sa^*}[j_0])$ is induced from $\mathsf{suf}(sa^*[i_1])$ and $\mathsf{suf}(sa^*[j_1])$, where $\mathsf{sub}(\overline{sa^*}[i_0], sa^*[i_1])$ and $\mathsf{sub}(\overline{sa^*}[j_0], sa^*[j_1])$ are two LMS-substrings, 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.

Given condition (2), the two sets of suffixes induced from any two LMS suffixes will not overlap (explain this...), hence the sa produced by the inducing processing must be a permutation of all suffixes.

□

The first condition of Theorem **??** is naturally satisfied when computing sa^* from sa_1. A naive solution for checking the second condition is to keep a copy of $\overline{sa^*}$ when computing sa from sa^* and compare it with sa^* afterward. This takes linear time, space and I/O overhead. We prefer to check equality of them by comparing their fingerprints, which can be calculated during the scan of these arrays in S1″ and S3″.

Corollary 4.6. $sa[0,n)$ is the SA for the input string $x[0,n)$ with a high probability if the following conditions are satisfied for all $i \in [0, n_1)$:
(1) $sa^*[i]$ is a permutation of all the S*-type suffixes and $sa^*[0] = n-1$.
(2) The fingerprints of sa^* and $\overline{sa^*}$ are equal.

As can be seen in Fig. 1, both sa^* and $\overline{sa^*}$ contain items $\{14, 8, 5, 2\}$ arranged in the same order. By using the fingerprinting function introduced in the next subsection, the fingerprints can be calculated in linear time using constant space. This implies another probabilistic algorithm designed by Method B.

4.3 Fingerprinting Function

Methods A and B check equality of two integer arrays by computing and comparing their fingerprints. In our programs, we select the Karp-Rabin fingerprinting function to compute the fingerprints in need. As depicted in Fig. 2,

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.

the fingerprint of A is calculated according to Formulas 4.7-4.8, where L is a prime and δ is an integer randomly chosen from $[1, L)$. It should be noticed that two equal arrays must have an identical fingerprint, but the inverse is not always true. Fortunately, the error probability can be reduced to a negligible level by setting L to a large value.

Formula 4.7. $\mathsf{fp}(A[0,-1]) = 0$.

Formula 4.8. $\mathsf{fp}(A[0,i]) = \mathsf{fp}(A[0,i-1]) \cdot \delta + A[i] \mod L$ for $i \geq 0$.

5 Experiments

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

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

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+. 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.

According to Corollary 4.6, Method A must check both sa^* and sa to accomplish verification. Similar to Method B, the overhead for checking sa is mainly caused by fingerprint calculations and thus can be neglected. On the other hand, we can apply the method proposed in [11] to ensure the correctness of sa^* within sorting complexity, where the time and space in need is proportional to the number of S*-type characters in x. Because the ratio of S*-type characters to all in x is commonly one-third in real-world datasets, the checking process for sa^* will not become the bottleneck for Method A. To achieve a higher speed, we can also parallelize the checking processes for sa and sa^* using more computing resources.

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

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.

6 Conclusion

In this paper, we proposed two methods that enable any IS suffix sorting algorithm to build and check SA simultaneously. In particular, the probabilistic algorithm designed by the second method is rather lightweight in terms of that it takes negligible time and space compared with the existing IS suffix sorting and checking algorithms. We also made an attempt to improve the space performance of DSA-IS using new substring sorting and naming methods. The experimental results shows that our program for the adapted algorithm DSA-IS+ runs as fast as that for eSAIS and requires only half as much disk space as the latter on various real-world datasets. We also showed that the space

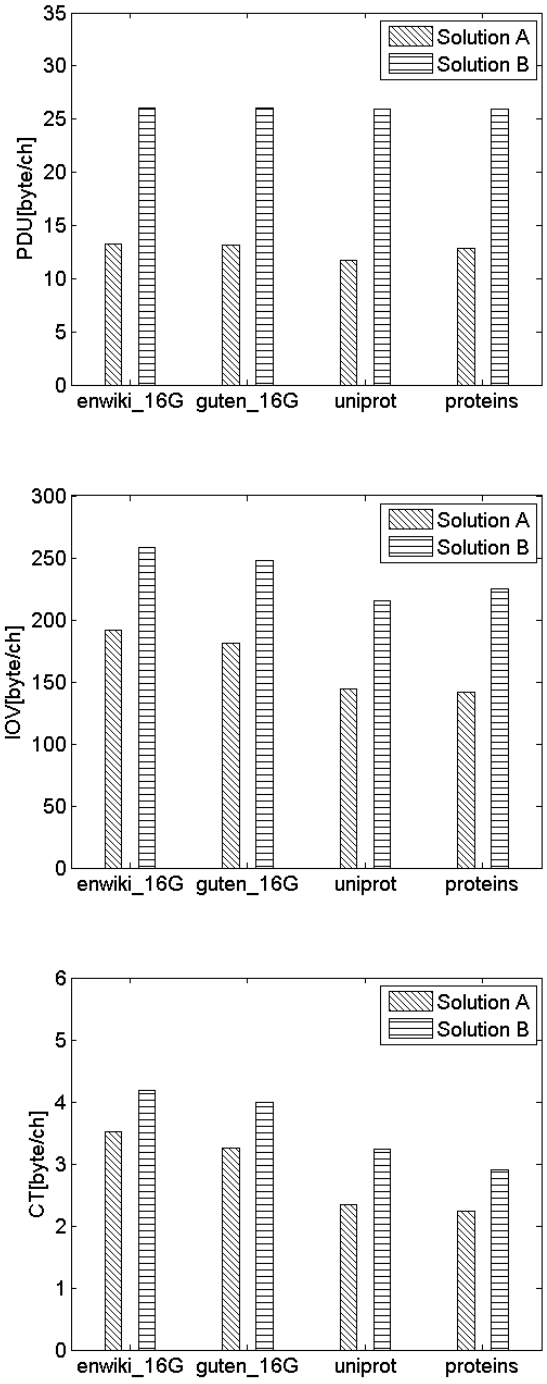


Fig. 4. A comparison of Solutions A and B on various corpora in terms of peak disk usage, I/O volume and construction time, where $D = 4$.

performance of this program can be further optimized using some optimization techniques. In our next paper, we will propose another disk-based suffix sorting algorithm that takes only $1n$ work space to run.

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.

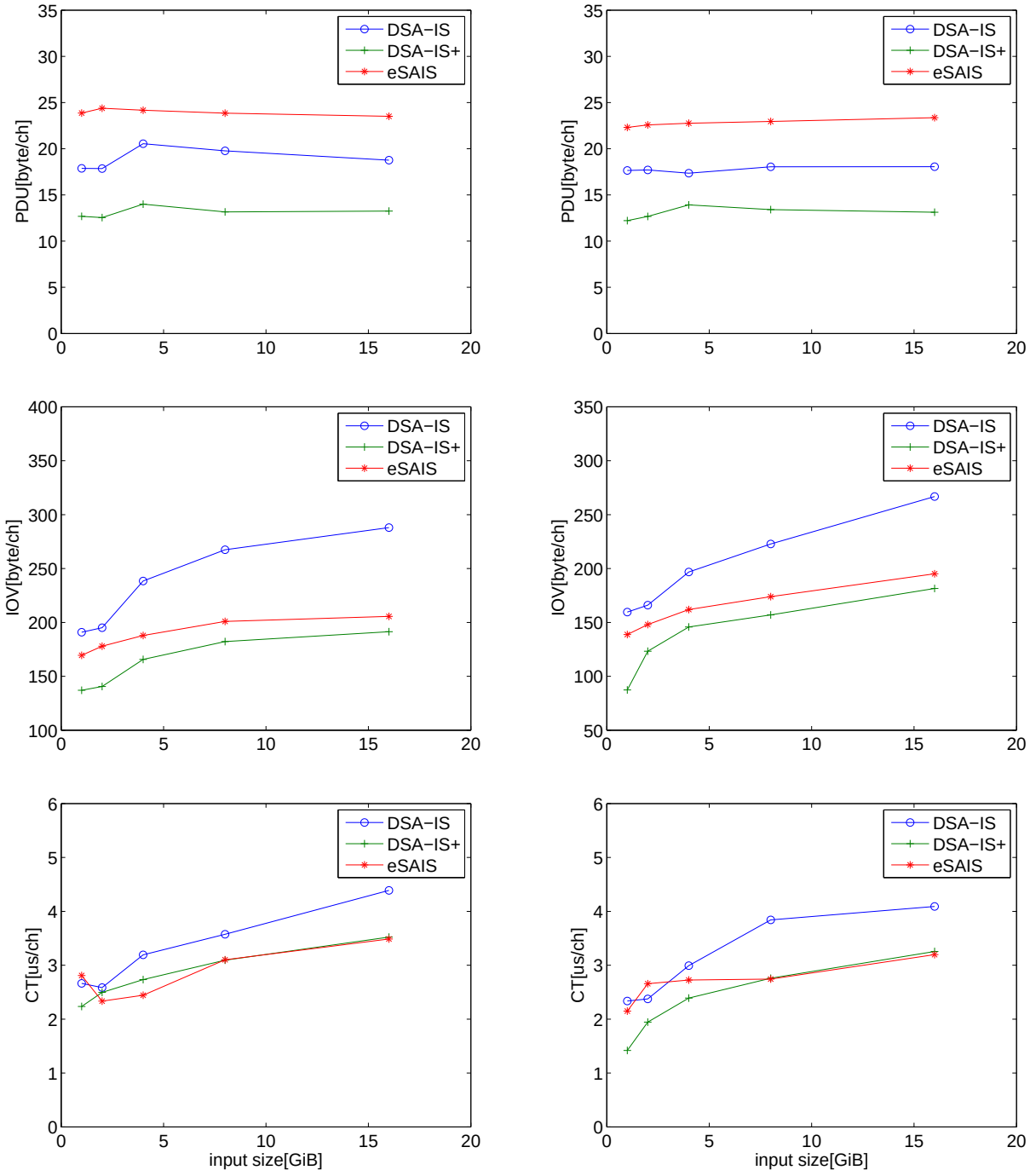
[8] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel External Memory Suffix Sorting," in *In proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching*, Ischia Island, Italy, July 2015, pp. 329–342.

[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.