Building and Checking Suffix Array Using Induced Sorting Method

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

Abstract—The induced sorting (IS) method has been applied to designing efficient suffix array (SA) construction algorithms on both internal and external memory models. In this paper, we propose two checking methods that enable any IS suffix sorters to check an SA when it is being built. The first method is capable of checking both suffix and LCP arrays for strings on constant alphabets, while the second method is specific for checking SA only but applicable to strings on both constant and integer alphabets. For performance analysis, we integrate the second checking method into the existing SA construction algorithm DSA-IS, where the design of DSA-IS is adapted by new substring sorting and naming methods for reducing its peak disk use to at most 14 times as input size. From our experimental results, the time, space and I/O consumptions for verification by our second method is negligible to that for construction by the adapted DSA-IS. This convinced that our checkers, together with the state-of-the-art IS suffix sorters, may constitute efficient solutions for building and checking SA at the same time.

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

1 Introduction

The suffix array of any constant or integer string can be built in linear time and RAM space by SA-IS [1]. The induced-sorting method has been also applied to designing fast external-memory SA construction algorithms eSAIS [2], DSA-IS [3] and SAIS-PQ [4]. These three disk-based variants have better time complexities than the other alternatives (e.g., DC3 [5], bwt-disk [6], SAscan [7] and pSAscan [8]), but they all suffer from a bottleneck due to the large disk space in use. For example, when building SA encoded by 40-bit integers, the peak disk use for eSAIS, DSA-IS and SAIS-PQ are $24n$, $20n$ and $15n$, respectively. This situation changed due to the work presented in [9], which proposed a carefully engineered IS suffix sorter fSAIS that takes no more than $8n$ disk memory for any constant or integer string of size $n \leq 2^{40}$ [9]. As reported, fSAIS reuses the substring/suffix sorting and naming methods of DSA-IS and SAIS-PQ, it achieves the remarkable space performance by means of a monotone priority queue for sorting fixed-size tuples and an I/O layer for allocating and deallocating disk space in a fine-grained way. The successful design of fSAIS shows a great potential for sharply decreasing the disk space requirements of the current programs for DSA-IS and SAIS-PQ. In this paper, we present our first attempt to improve DSA-IS by new substring sorting and naming methods. From the experiments, our program for the adapted algorithm DSA-IS+ takes $12n$ disk space in average and runs as fast as eSAIS on various real-world datasets. We will demonstrate that this program can be further improved using the optimization technique for implementing fSAIS.

To avoid computation errors caused by implementation bugs and/or hardware malfunctions, an SA should be checked to ensure its correctness after construction. Currently, the software packages for DC3 and eSAIS provide users a checker based on the method proposed in [5], which performs two disk sorts for $\mathcal{O}(n)$ fixed-size tuples with an integer key. When running in external memory, this checker takes one-third as much time and I/O consumptions as the programs for DSA-IS and eSAIS, it also requires nearly the same disk space as the two SA builders. Against the background, we propose two checking methods that enable any IS suffix sorting algorithm to build and check SA at the same time. In comparison with the previous checking method, they are rather lightweight in terms of time, space and I/O volume. It should be noticed that these two methods are designed for different situations. In brief, the first method can check both suffix and LCP arrays, but it is specific for strings on constant alphabets. The second method can be applied to strings on constant and integer alphabets, but it can check SA only. As will be seen in our experiments, the consumption for checking by our second method is negligible compared with that for building by DSA-IS+.

The rest of this paper is organized as follows. Section 2 describes the framework of IS suffix sorting algorithms and the idea of new substring sorting and naming methods, Section 3 the designs of two checking methods, Sections 4 the experimental results, and Section 5 the concluding remarks.

- *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.*

2 Builder

2.1 Preliminaries

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

Character/substring/suffix classification. 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. Furthermore, 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. An array t records the type of all the characters in x, $t[i]$ is set to 1 or 0 if $x[i]$ is S-type or L-type, respectively. Each substring/suffix is of the same type as its heading character. In this paper, we only consider the substrings ending with an S*-type character.

Predecessor and successor. Given two characters $x[i]$ and $x[i + 1]$, $x[i]$ 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 two suffixes or substrings starting with neighboring characters.

Reduced string. Split x into multiple substrings such that each substring only contains two S*-type characters at the starting and ending positions. Assign each substring a name and replace them with their names to produce a reduced string x_1, where the name indicates the lexical order of the corresponding substring among all.

Suffix array. The suffix array sa indicates the lexicographical 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 SA for x_1 by sa_1.

2.2 Introduction to IS Suffix Sorting Algorithms

Algorithm 1: The general framework for an IS suffix sorting algorithm.

```
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.
```

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 sa from these sorted suffixes. According to the IS principle, the order of two substrings/suffixes is determined by comparing their heading characters and successors in sequence during the execution of reduction/induction phase. Notice that this key operation takes multiple random accesses to x and sa. It can be done very quickly if both x and sa are wholly loaded into RAM; otherwise, each access may take an individual I/O operation to external memory and thus becomes the bottleneck.

To amortize the I/O overhead, eSAIS, DSA-IS and SAIS-PQ use different methods to obtain heading characters in a disk-friendly way. Particularly, DSA-IS first divides x into multiple blocks and calls SA-IS to sort suffixes in each block at the beginning of induction phase, where the heading characters in need are written to external memory in their access order. Then, it organizes the sorted suffixes in a priority queue and visits them sequentially to induce the order of their predecessors following the IS principle. Because the required heading characters was already stored on disks, they can be read into RAM by sequential I/O operations when needed. This technique is also applied to sorting substrings during the reduction phase. Afterward, a naming process is conducted to check equality of lexicographically neighboring substrings for producing the reduced string x_1. To do this, DSA-IS maintains a copy for each substring in external memory and arranges them in their sorted order. As can be seen in Section 4, our program for DSA-IS requires less disk space than that for eSAIS, but it runs slower than the latter due to the large I/O volume for sorting and naming S*-type substrings.

2.3 Improvements on DSA-IS

We first describe a new method for sorting S*-type substrings during the reduction phase. Given an integer value D, all the S*-type substrings are classified into long and short categories with respect to whether or not they contain more than D characters. Our sorting method executes the following three steps in sequence after splitting x into blocks $\{b_1, b_2, ...\}$.

S1 Sort S*-type substrings in each block by calling the reduction phase of SA-IS. Copy the short substrings and store them on the disk in their sorted order. Copy the heading characters needed for sorting long substrings and store them on the disk in their access order.

S2 Sort long S*-type substrings by the same way of DSA-IS. Copy the leftmost D characters of the long substrings and store them on the disk in their sorted order.

S3 Merge short and long S*-type substrings by a multi-way sorter.

Assume the number of blocks is k, S1 copies the sorted short substrings in b_i to an external memory vector vs_i and S2 copies the sorted long substrings to an external memory vector vl. Then, S3 uses a sorter to merge these partial results, where the sorter maintains a minimal heap in RAM to determine the current smallest substring in $vs_1, vs_2, ..., vs_k$ and vl. For any two substrings in the heap, their lexicographical order can be determined in $\mathcal{O}(D)$ time by literally comparing their characters from left to right. This sorting method achieves a good performance when most of the S*-type substrings are short. This makes it feasible in practice because the average length of S*-type substrings in real-world datasets are typically small.

Next, we describe how to name S*-type substrings at the same time when they are being sorted. Given two S*-type substrings successively popped from the minimal heap, we make a literal comparison of them to check their equality if they are both short; otherwise, we use the naming method of SAIS-PQ to determine their equality in S2. Specifically, substrings are sorted by comparing their heading characters and the names of their sorted successors in S2, where two substrings are of the same name if and only if their heading characters and the names of their successors are both equal. By recording the names of long S*-type substrings in their sorted order, it takes constant time to check equality of two lexicographically neighboring long substrings in S3.

2.4 Discussion

Section 4 shows that, by using the new substring sorting and naming methods, our program for the enhanced DSA-IS has an advantage over that for eSAIS in terms of space and I/O volume, but its average peak disk use is still higher than that for fSAIS by around $5n$. This is mainly due to the use of external-memory containers (vectors, sorters and priority queues) provided by the STXXL library [10]. More specifically, the STXXL's containers fail to free the disk space for saving temporary data immediately after they are no longer to use. This implementation issue can be solved by storing temporary 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 similar to that for fSAIS.

3 Checkers

3.1 Prior Art

The widespread software packages for DC3 and eSAIS provide users a checker to perform verification. Theorem 3.1 gives the main idea behind this checker.

Theorem 3.1. $sa[0,n)$ 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 indicates the relative order of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ among all the suffixes, respectively.

Proof: The proof is available in [5]. □

This checker performs two passes of integer sorts and each sort involves n fixed-size tuples. When implemented in external memory, it requires large disk space and consumes high I/O volume, resulting in a non-negligible checking overhead. As will be seen in Section 4, the peak disk use for checking an SA encoded by 40-bit integers is around $21n$, while the I/O volume is no less than $50n$. In this section, we describe two lightweight SA checking methods based on the IS principle. Both of them can be seamlessly integrated into any IS suffix sorting algorithm to build and check SA simultaneously.

3.2 Method A

3.2.1 Preliminaries

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

Suffix and LCP buckets. All the suffixes in sa are naturally partitioned into multiple buckets. Each bucket contains suffixes starting with a same character and occupies a contiguous interval of sa. For example, $\mathsf{sa\_bkt}(c_0)$ gathers all the suffixes starting with c_0. This bucket can be further partitioned into two parts, where the left and right parts only contain the L-type and S-type suffixes, respectively. For short, we denote the left and right parts by $\mathsf{sa\_bkt_L}(c_0)$ and $\mathsf{sa\_bkt_S}(c_0)$, respectively. Accordingly, lcp can be also partitioned into multiple buckets and $\mathsf{lcp\_bkt}(c_0)$ records the LCP-values of suffixes in $\mathsf{sa\_bkt}(c_0)$. Similarly, we define $\mathsf{lcp\_bkt_L}(c_0)$ and $\mathsf{lcp\_bkt_S}(c_0)$ for $\mathsf{sa\_bkt_L}(c_0)$ and $\mathsf{sa\_bkt_S}(c_0)$, respectively.

Suffix and LCP arrays for S-type suffixes.* 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, respectively. The i-th element in sa^* records the starting position of the $(i+1)$-th smallest S*-type substring, while the i-th element in $lcp^*[i]$ records the LCP-value of $\mathsf{suf}(sa^*[i])$ and $\mathsf{suf}(sa^*[i-1])$.

3.2.2 Idea

During the induction phase, SA-IS first computes sa^* by sorting the starting positions of all the S*-type suffixes with their ranks indicated by sa_1 and then induces sa from sa^* by S1′-S3′.

S1′ Scan sa^* rightward with i decreasing from $n_1 - 1$ to 0. For each scanned item $sa^*[i]$, insert it into the rightmost empty position of $\mathsf{sa\_bkt_S}(x[sa^*[i]])$.
S2′ 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 $t[sa[i]-1] = 0$.
S3′ 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 $t[sa[i]-1] = 1$.

An example of the above inducing process is given in Fig. 1. In line 6, we find the end of each bucket and insert the sorted S*-type suffixes into the corresponding buckets. Then, we find the head of each bucket (marked by the symbol $\wedge$) and scan sa rightward for inducing the order of L-type suffixes. In lines 8-9, we check $\mathsf{suf}(14)$ (marked by the symbol @) and find its predecessor $\mathsf{suf}(13)$ is L-type, thus 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. After that, we find the end of each bucket and scan sa leftward for inducing the order of S-type suffixes in lines 18-29. When scanning $sa[14]$, we see $x[3]$ is S-type and thus put $\mathsf{suf}(3)$ into the current rightmost empty position in $\mathsf{sa\_bkt_S}(i)$. Following the same, we get all the S-type suffixes sorted in line 29.

As stated below, S1′-S3′ also constitute the sufficient and necessary conditions for a correct sa.

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^*.

Lemma 3.2. For any IS suffix sorting algorithm, its output $sa[0,n)$ 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 SS1-SS3.

Suppose sa^* is correct[1], this lemma suggests a method to check sa when it is induced from sa^* during the induction phase. Specifically, in S2′ and S3′, each item induced into sa will be scanned later to induce the order of its predecessor. If the induced and scanned values for each item are equal, then sa is correct according to Lemma 3.2. The problem to be solved here is that when an item is induced into a bucket, its value in sa will not be scanned at once. Our solution for this is to integrate the building and checking processes into a whole, by ensuring the sequence of items induced into a bucket is identical to that scanned later. For the purpose, we increasingly compute the fingerprints of both sequences and check their equality at the end of the induction phase. If the two fingerprints for each bucket are equal, then the second condition of Lemma 3.2 will be seen with a high probability. As a result, sa can be built and probabilistically checked at the same time, for this we have Corollary 3.3.

1. The correctness of sa^* can be checked by a sparse SA checker, such as [11]. The checking processes for sa^* and sa can be executed in parallel.

Corollary 3.3. $sa[0,n)$ is the SA for the input string $x[0,n)$ with a high probability if the following conditions are satisfied for all $c \in \Sigma$:
(1) sa^* is correct.
(2) The fingerprints of $\mathsf{sa_{I1}}(c)$ and $\mathsf{sa_{S1}}(c)$ are equal.
(3) The fingerprints of $\mathsf{sa_{I2}}(c)$ and $\mathsf{sa_{S2}}(c)$ are equal.

Notice that $\mathsf{sa_{I1}}(c)$ and $\mathsf{sa_{I2}}(c)$ 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 $\mathsf{sa\_bkt_S}$.

3.3 Method B

We introduce Lemma 3.4 before the presentation of Method B. This statement has been applied to deriving Theorem 3.1 for comparing two suffixes given the order of their successors already known.

Lemma 3.4. For $i,j \in [0,n)$ and $i \neq j$, $\mathsf{suf}(i) < \mathsf{suf}(j) \Leftrightarrow (x[i], \mathsf{suf}(i+1)) < (x[j], \mathsf{suf}(j+1))$.

Below we describe the idea of Method B and prove its correctness using Lemma 3.4 and Theorem 3.1, where $\overline{sa^*}$ is retrieved from sa and it records the starting positions of all the S*-type suffixes in their sorted order.

Theorem 3.5. For any IS suffix sorting algorithm, its output $sa[0,n)$ 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 the starting position of an S*-type suffix and it differs from any other items in sa^*.
(2) $sa^*[i] = \overline{sa^*}[i]$.

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

Suppose sa is not a permutation of $[0,n)$. Without loss of generality, let $i \neq j$ and $sa[i] = sa[j]$, then $u \neq v$ and $sa[u] = sa[v]$ according Theorem 3.1, where $\mathsf{suf}(sa[u])$ and $\mathsf{suf}(sa[v])$ are the successors of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$, respectively. By recursively replacing (i,j) with (u,v), we can find two items $sa[i_0]$ and $sa[j_0]$ storing the starting position of the same S*-type suffix, indicating that sa^* contains duplicate items. This violates condition (1) of Theorem 3.5.

Suppose $i < j$ and $\mathsf{suf}(sa[i]) > \mathsf{suf}(sa[j])$. If $x[sa[i]] < x[sa[j]]$, then $\mathsf{suf}(sa[i]) < \mathsf{suf}(sa[j])$ according to Lemma 3.4; otherwise, if $x[sa[i]] > x[sa[j]]$, then $i > j$ according to Theorem 3.1. Both two conditions lead to a contradiction. Therefore, we must have $x[sa[i]] = x[sa[j]]$. If so, then $u < v$ and $\mathsf{suf}(u) > \mathsf{suf}(v)$, where $\mathsf{suf}(sa[u])$ and $\mathsf{suf}(sa[v])$ are the successors of $\mathsf{suf}(sa[i])$ and $\mathsf{suf}(sa[j])$, respectively. By recursively replacing (i,j) with (u,v), we can find two elements $sa[i_0]$ and $sa[j_0]$ such that at least one stores the starting position of an S*-type suffix. We continue the proof following one of the three cases below:

(1) $x[sa[i_0]]$ is S*-type and $x[sa[j_0]]$ is L-type. In this case, $x[sa[i_0]{+}1] > x[sa[j_0]{+}1]$. Therefore, $u_0 > v_0$ according to Theorem 3.1. This indicates that $(x[sa[i_0]], u_0) > (x[sa[j_0]], v_0)$ and thus $i_0 > j_0$ according to Theorem 3.1. This leads to a contradiction.
(2) $x[sa[i_0]]$ is L-type and $x[sa[j_0]]$ is S*-type. In this case, $x[sa[i_0]+1] < x[sa[j_0]+1]$. Therefore, $\mathsf{suf}(sa[u_0]) <$

$\mathsf{suf}(sa[v_0])$ according to Lemma 3.4. This indicates that $(x[sa[i_0]], \mathsf{suf}(sa[u_0])) < (x[sa[j_0]], \mathsf{suf}(sa[v_0])$ and thus $\mathsf{suf}(sa[i_0]) < \mathsf{suf}(sa[j_0])$ according to Lemma 3.4. This leads to a contradiction.

(3) $x[sa[i_0]]$ and $x[sa[j_0]]$ are both S*-type. Let (i_0, j_0) be the pair that maximize $sa[i_0] + sa[j_0]$ such that $i_0 < j_0$ and $\mathsf{suf}(sa[i_0]) > \mathsf{suf}(sa[j_0])$, then $u_0 > v_0$ and $\mathsf{suf}(sa[u_0]) > \mathsf{suf}(sa[v_0])$. 2 Because $x[i_0, u_0) = x[j_0, v_0)$, there must be $i_0 > j_0$ according to Theorem 3.1, thus sa^* and $\overline{sa^*}$ differ from each other. This violates condition (2) of Theorem 3.5.

□

The first condition of Theorem 3.5 is naturally satisfied when computing sa^* from sa_1. A naive method 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 3.6. $sa[0, n)$ 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 the starting position of an S*-type suffix and it differs from any other items in sa^*.
(2) The fingerprints of sa^* and $\overline{sa^*}$ are equal.

In Fig. 1, both sa^* and $\overline{sa^*}$ contain 4 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.

3.4 Discussion

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

03 compute $\mathrm{fp}(A[0,p])$ with $L = 23$, $\delta = 5$:

04 $\mathrm{fp}(A[0,0]) = \mathrm{fp}(A[0,-1]) * 5 + A[0] \mod 23 = 14$

05 $\mathrm{fp}(A[0,1]) = \mathrm{fp}(A[0,0]) * 5 + A[1] \mod 23 = 9$

06 $\mathrm{fp}(A[0,2]) = \mathrm{fp}(A[0,1]) * 5 + A[2] \mod 23 = 4$

07 $\mathrm{fp}(A[0,3]) = \mathrm{fp}(A[0,2]) * 5 + A[3] \mod 23 = 22$

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

Methods A and B check equality of two integer arrays by comparing their fingerprints. In our programs, we choose to use the Karp-Rabin fingerprinting function to compute the fingerprints in need. As depicted in Fig. 2, the fingerprint of A is calculated according to Formulas 3.7-3.8, 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 probability of a false match can be reduced to a negligible level if L is set to a large value.

2. Notice that (u_0, v_0) must exist because the sentinel is a size-one S*-type substring smaller than others.

Formula 3.7. $\mathsf{fp}(A[0, -1]) = 0$.

Formula 3.8. $\mathsf{fp}(A[0, i]) = \mathsf{fp}(A[0, i-1]) \cdot \delta + A[i] \mod L$ for $i \geq 0$.

We also point out that Method A can check lcp when it is being built from lcp^* following the IS principle [12]. This is done by comparing the sequence of items induced into an lcp bucket with that scanned from the same bucket. If the two sequences of each bucket are equal, then lcp is correctly the LCP array for sa.

4 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.

4.1 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

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

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. Therefore,the proposed substring sorting and naming methods are practical for real-world datasets.

4.2 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 3.6, 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. We also point out that this checking process can be parallelized with that for sa to achieve a higher speed.

5 Conclusion

In this paper, we made an attempt to improve the space performance of DSA-IS by new substring sorting and naming methods. For implementation convenience, we currently employ the STXXL's containers to perform reading, writing and sorting on the disk. The experimental results shows that our program for the adapted algorithm DSA-IS+ runs as fast as eSAIS and requires only half disk space as that for the latter on various real-world datasets. This program can be further optimized to approach the optimal space performance by means of the external-memory vector, sorter and priority queues supporting fine-grained disk space allocation and deallocation.

We also proposed two methods that enable any IS suffix sorting algorithm to build and check SA simultaneously. The second method is rather lightweight in terms of that its time and space complexities are negligible compared with that of the existing IS suffix sorting/checking algorithms. We will describe in another paper a disk-based suffix sorting algorithm taking only $1n$ work space. By augmenting with this new suffix array builder, the proposed checking methods may potentially constitute the best solution for the situations where checking is a must after building.

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. 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.

[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.

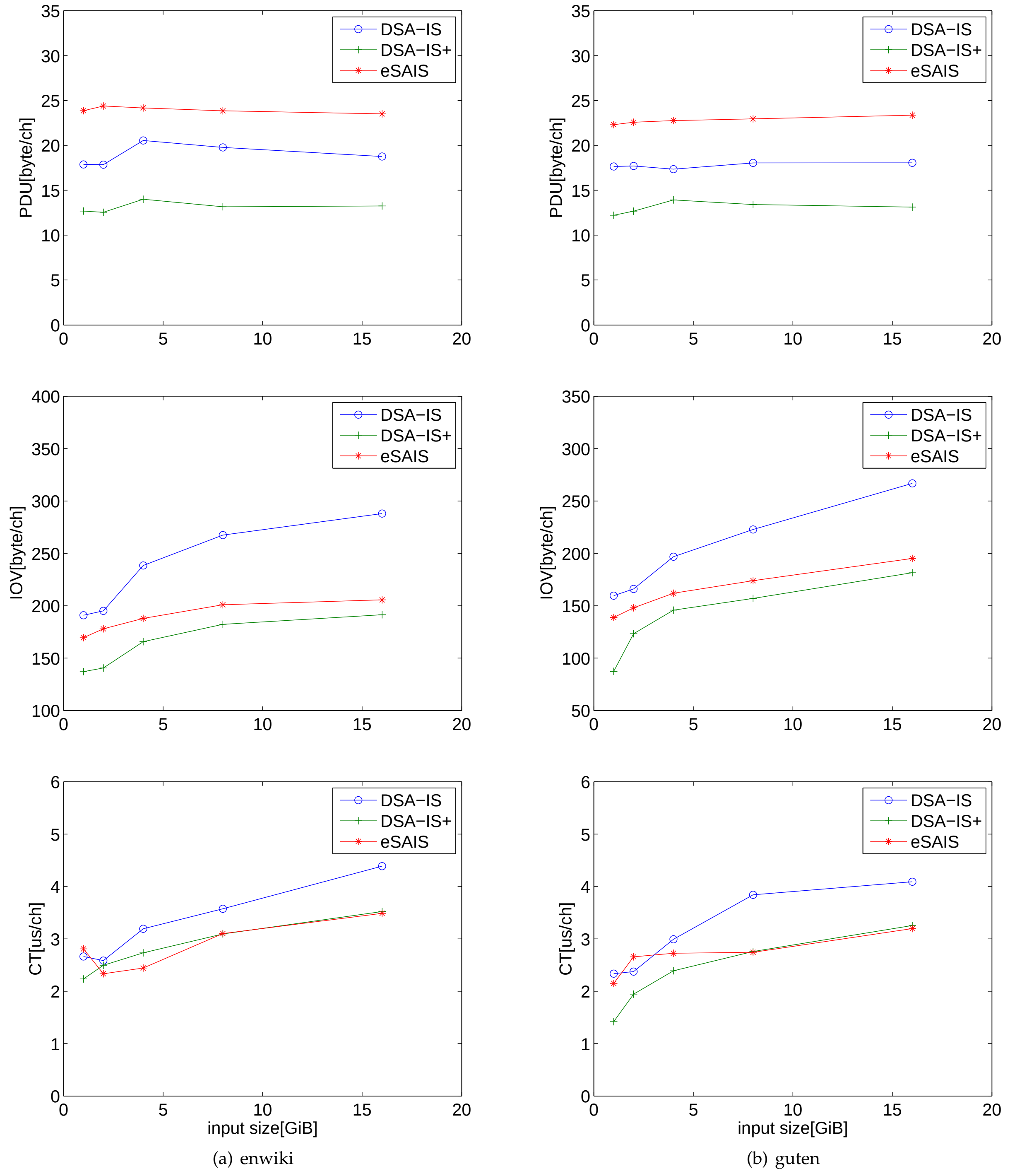

Fig. 3. A comparison of DSA-IS, DSA-IS+ and eSAIS on guten and enwiki in terms of peak disk usage, I/O volume and construction time, where $D = 4$ and the input size varies in {1, 2, 4, 8, 16} GiB.

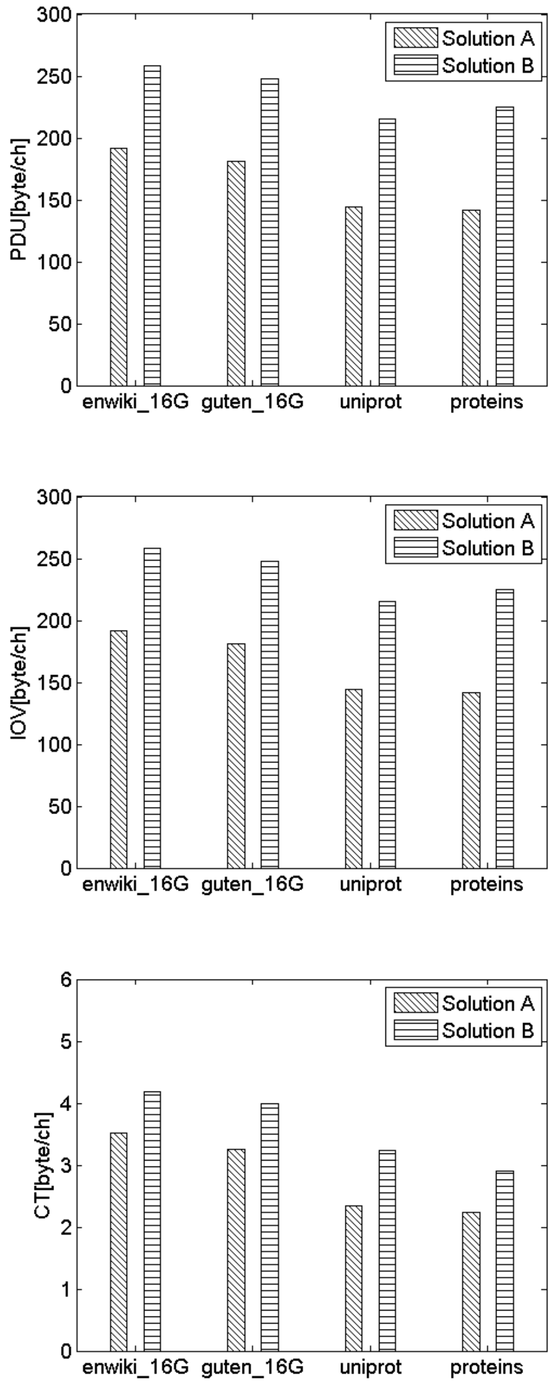

Fig. 4. A comparison of DSA-IS and eSAIS on various corpora in terms of peak disk usage, I/O volume and construction time.