Building and Checking a Suffix Array by Induced Sorting

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

Abstract—A suffix array (SA) can be built on both internal and external memory models following the induced sorting (IS) principle. We propose two methods for checking an SA when it is being built by any IS suffix sorter. The first method can be used to probabilistically check both suffix and longest common prefix (LCP) arrays, while the second method is for checking SA deterministically. To achieve a good performance, we combine the Karp-Rarbin fingerprinting techinque into our methods to 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 within linear time and constant RAM space. We integrate the algorithm designed by the second method into the existing disk-based suffix sorting algorithm DSA-IS for evaluation, where the experimental results indicate that the checking overhead is considerably smaller than the building consumption.

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

1 Introduction

The SA-IS algorithm takes linear time and RAM space to build an SA on internal memory model [1]. The IS method determines the lexical order of two suffixes by comparing their heading characters and successors in sequence, where the lexical order of their successors is determined recursively following the same way. Recently, the IS method has been applied to designing three disk-based suffix sorting algorithms eSAIS [2], DSA-IS [3] and SAIS-PQ [4]. In particular, eSAIS can build both suffix and LCP arrays at the same time. These IS algorithms have linear I/O complexity better than the others designed by different methods (e.g., DC3 [5], bwt-disk [6], SAscan [7] and pSAscan [8]), but they 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. As reported, the average disk use for pSAscan to build a size-n SA encoded by 40-bit integers is around $7.5n$ bytes, while that for eSAIS, DSA-IS and SAIS-PQ are $24n$, $18n$ and $15n$ bytes, respectively. The poor space performance for these IS algorithms is mainly because their current programs fail to 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 into multiple files and deleting a file immediately when the data in it is not needed anymore. This trick has been employed by fSAIS [9] to engineer the external-memory IS method, where the peak disk use is $8n$ bytes for building an SA of 40-bit integers in the given experiments.

- *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.*

While the research on developing various SA builders using the IS method is evolving, 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. We propose in this paper two methods that enable the IS method to build and check an SA at the same time. Our first method is probabilistic and can be extended to check both suffix and LCP arrays, while the second method is deterministic and designed for checking SA only. By combining the Karp-Rabin fingerprinting technique [10], we design two probabilistic algorithms to perform checking correctly with a negligible error probability. The key operation of both algorithms is to check equality of two integer sequences by computing and comparing their fingerprints. As will be shown, the algorithm designed by the first method checks an SA within a integer sorting complexity and the algorithm designed by the second method checks an SA in linear time. For a further study, we first use new substring sorting and naming methods to improve the design of DSA-IS and then integrate the algorithm designed by the second method into the adapted DSA-IS for evaluation. Our experimental results indicate that the time, space and I/O volume for checking SA is negligible in comparison with 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 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. In the rest of 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 $x[i]$. 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 represents the rank of this substring among all. We interchangeably use "rank" and "name" to indicate the lexical order of a substring.

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 $\mathsf{sa\_bkt_S}(c_0)$ that contain L-type and S-type suffixes, respectively. Similarly, we 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

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.

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 duplicate 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.
```

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 may take an individual I/O operation to disk, leading to a performance degradation. The DSA-IS algorithm solves this problem by executing the following two steps during a reduction/induction phase:

S1 Split x into blocks and sort substrings/suffixes in 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 in x according to the lexical order of their heading characters and the ranks of their successors by using an external-memory heap.

S2 induces the lexical order of substrings/suffixes by using an external-memory heap. It continually pops the top substring/suffix from the heap and induces its predecessors into the heap according to the rank of the top substring/suffix and the heading character of the predecessors. For all the induced substrings/suffixes, their heading characters are already arranged in access order on disk by S1 and thus can be retrieved from external memory by sequential I/O operations before use. 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 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′ Reuse S1 to sort all the substrings in each block. Copy the heading characters of the induced long substrings to external memory in access order and copy the induced short S*-type substrings to external memory in sorted order.

S2′ Reuse S2 to sort the long substrings in x. Copy the leftmost D characters of the long S*-type substrings to external memory in sorted order.

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

The short S*-type substrings in each block are already sorted and organized as a sequence in external memory after S1′, while the long S*-type substrings in x are already sorted and 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 performs a literal string comparison to compare any two S*-type substrings in $\mathcal{O}(D)$[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 by S1′-S3′. The key operation is to check equality of two substrings successively popped from the heap in S3′. If either of the substrings is short, then we literally compare them in $\mathcal{O}(D)$ time. Otherwise, we check whether their ranks are equal by comparing their heading characters and the ranks of their successors in S2′. For the same purpose, SAIS-PQ also applies a similar technique to merge the substring sorting and naming processes. Our experiments in Section 5 show 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.

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 following the induced sorting principle. □

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 conditions for checking SA.

Lemma 4.2. For any IS suffix sorter, its output $sa[0, n)$ is the SA for $x[0, n)$ if the following conditions are satisfied at

```
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    s̅a̅*̅: 14          8   5   2
```

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

the top recursion level:
(1) sa^* is correctly computed.
(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.

Suppose $sa[i] = sa[j]$ and $i \neq j$, because S1″-S4″ are correctly implemented, $sa[p] = sa[q]$ 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 correctly computed, leading to a contradiction. This satisfies the first condition of Lemma 4.1.

□

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. These methods adopt different ways to ensure the correctness of sa^*. Particularly, our first method can be used to check both suffix and LCP arrays.

4.2.1 Method A

The induced sorting method can be applied to building LCP array as well [11]. That is, the LCP-value of two substrings can be induced from that of their successors as following: the LCP-value is equal to zero if the two substrings have different heading characters; otherwise, the LCP-value is one greater than that of their successors. Assume the induction phase for inducing sa and lcp from sa^* and lcp^* is correctly implemented at the top recursion level, we show in Lemma 4.3 a set of conditions for checking both sa and lcp when they are built by an IS builder like [2], [11].

Lemma 4.3. Assume the induction phase at the top recursion level is correctly implemented, the output $sa[0, n)$ and $lcp[0, n)$ of an IS builder are the suffix and LCP arrays for the input string $x[0, n)$ with a high probability if the following conditions are satisfied:
(1) sa^* and lcp^* are correct according to the checking result of the sparse suffix and LCP arrays probabilistic checker [12].
(2) $sa[i]$ and $lcp[i]$ are equal to the values induced into the corresponding positions of sa and lcp during the induction phase, for $i \in [0, n)$.

The checker in [12] verifies sa^* and lcp^* within an integer sorting complexity. For each pair of neighboring suffixes in the given sa, it first employs the Karp-Rabin fingerprinting technique to compute the fingerprints of their leftmost ℓ characters, where ℓ is their LCP-value indicated by the given lcp. If the two fingerprints are equal, then it continues to check if the lexical order of their $(\ell + 1)$-th characters corresponds to that of themselves indicated by the given sa. The second condition in Lemma 4.3 is intended for detecting malfunctions other than implementation bugs. Because each suffix induced into sa/lcp will be latter scanned for inducing the order of its predecessor, this condition can be checked by determining whether the induced and scanned values for each suffix are equal. The problem 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 reuse the Karp-Rabin fingerprinting technique to increasingly compute the fingerprints of both sequences and check their equality in constant time at the end of 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 and lcp can be built and probabilistically checked at the same time.

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, $\mathsf{sa}_{\mathsf{I1}}(c)$ and $\mathsf{sa}_{\mathsf{I2}}(c)$ are two sequences respectively induced into $\mathsf{sa\_bkt_L}(c)$ and $\mathsf{sa\_bkt_S}(c)$, while $\mathsf{sa}_{\mathsf{S1}}(c)$ and $\mathsf{sa}_{\mathsf{S2}}(c)$ are two sequences respectively scanned from $\mathsf{sa\_bkt_L}(c)$ and sa_bkt$_{S}(c)$. The algorithm computes and compares the fingerprints of these sequences and those of $\mathsf{lcp}_{l1}(c)/\mathsf{lcp}_{l2}(c)$ and $\mathsf{lcp}_{S1}(c)/\mathsf{lcp}_{S2}(c)$ to check the second condition of Lemma 4.3.

4.2.2 *Method B*

Method B is deterministic and specific for checking SA only. It uses a different way to ensure the correctness of sa^*. Before our presentation, we first introduces 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 computed 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 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 $i \in [0, n_1)$.

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″. One way for deterministically checking the second condition is to copy both sa^* and $\overline{sa^*}$ to external memory and compare their elements one by one after the induction phase. Algorithm 3 applies a more space efficient way to checking equality of the two integer sequences with a negligible error probability. It reuses the Karp-Rabin fingerprinting technique to compute and compare the fingerprints of sa^* and $\overline{sa^*}$, where the fingerprint of sa^* is computed when placing the elements of sa^* into sa in S2″ and the fingerprint of $\overline{sa^*}$ is computed when inducing the S*-type suffixes into sa in S4″.

4.3 Fingerprinting Function

Both Algorithms 2 and 3 check equality of sequences by comparing their fingerprints. We can use any rolling hash function to compute the fingerprints in need. Notice that two equal sequences 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 techniques to compute these values 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 in this way, 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 fp(A[0,p]) with L = 23 , δ = 5:
04  fp(A[0,0]) = fp(A[0,−1]) ∗ 5 + A[0]  mod 23 = 14
05  fp(A[0,1]) = fp(A[0,0]) ∗ 5 + A[1]  mod 23 = 9
06  fp(A[0,2]) = fp(A[0,1]) ∗ 5 + A[2]  mod 23 = 4
07  fp(A[0,3]) = 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

Algorithms 2 and 3 take linear time and constant RAM space for computing the target sequences by the Karp-Rabin fingerprinting technique. Algorithm 2 also employs the checker presented in [12] to verify sa^* and lcp^* within an integer sorting complexity. Because at most one out of every two suffixes is S*-type and the checking process is only executed at the top recursion level, the overhead for checking sa^* and lcp^* is considerably smaller than that for constructing the two arrays by an IS builder.

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 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.

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 [12].
3     Compute the fingerprints of $\mathsf{sa}_{\mathsf{l1}}(c)$, $\mathsf{sa}_{\mathsf{S1}}(c)$, $\mathsf{lcp}_{\mathsf{l1}}(c)$ and $\mathsf{lcp}_{\mathsf{S1}}(c)$ when inducing the order and the LCP-values
        of L-type suffixes.
4     Check if $\mathsf{sa}_{\mathsf{l1}}(c) = \mathsf{sa}_{\mathsf{S1}}(c)$ and $\mathsf{lcp}_{\mathsf{l1}}(c) = \mathsf{lcp}_{\mathsf{S1}}(c)$.
5     Compute the fingerprints of $\mathsf{sa}_{\mathsf{l2}}(c)$, $\mathsf{sa}_{\mathsf{S2}}(c)$, $\mathsf{lcp}_{\mathsf{l2}}(c)$ and $\mathsf{lcp}_{\mathsf{S2}}(c)$ when inducing the order and the LCP-values
        of S-type suffixes.
6     Check if $\mathsf{sa}_{\mathsf{l2}}(c) = \mathsf{sa}_{\mathsf{S2}}(c)$ and $\mathsf{lcp}_{\mathsf{l2}}(c) = \mathsf{lcp}_{\mathsf{S2}}(c)$.
```

Algorithm 3: The Algorithm Based on Lemma 4.4.

```
1 Function CheckByMethodB($x, sa^*$)
2     Compute the fingerprints of $sa^*$ when inducing the order of L-type suffixes.
3     Compute the fingerprints of $\overline{sa^*}$ when inducing the order of S-type suffixes.
4     Check if $sa^* = \overline{sa^*}$ by comparing their fingerprints.
```

- 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.

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

6 Conclusion

By assuming the induction phase at the top recursion level is correctly implemented, we propose two methods that enable an IS builder to build and check an SA simultaneously. The probabilistic algorithm designed by Method B is rather lightweight, it takes negligible time and space to run compared to the existing IS suffix sorting and checking algorithms. We also made our first attempt to improve the performance of DSA-IS using new substring sorting and naming methods. Our 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] J. Fischer, "Inducing the LCP-array," in *In Workshop on Algorithms and Data Structures*, 2011, pp. 374–385.

[12] 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.

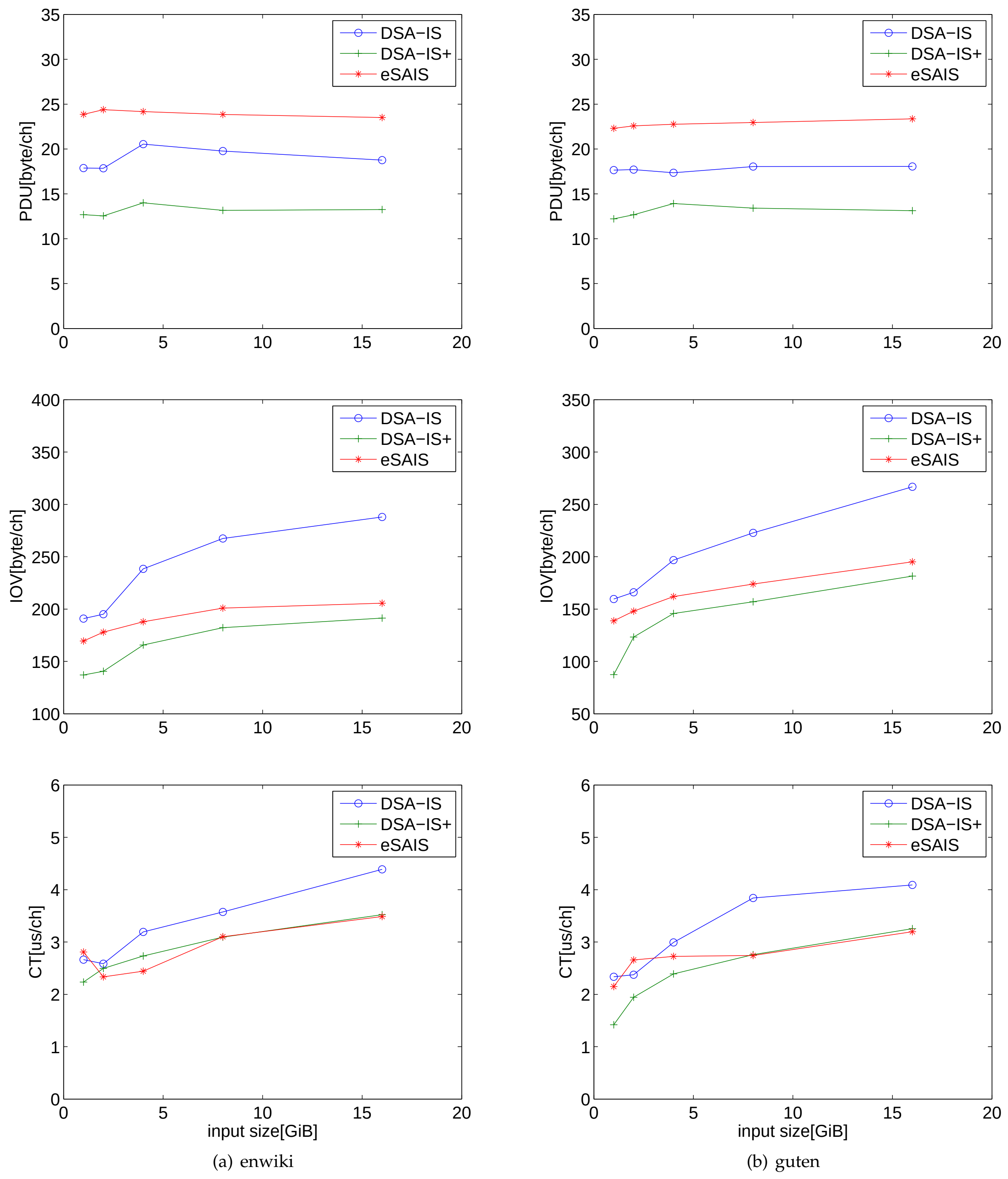

Fig. 3. A comparison of DSA-IS, DSA-IS+ and eSAIS on guten and enwiki in terms of peak disk usage, I/O volume and construction time, where $D = 4$ and the input size varies in {1, 2, 4, 8, 16} GiB.

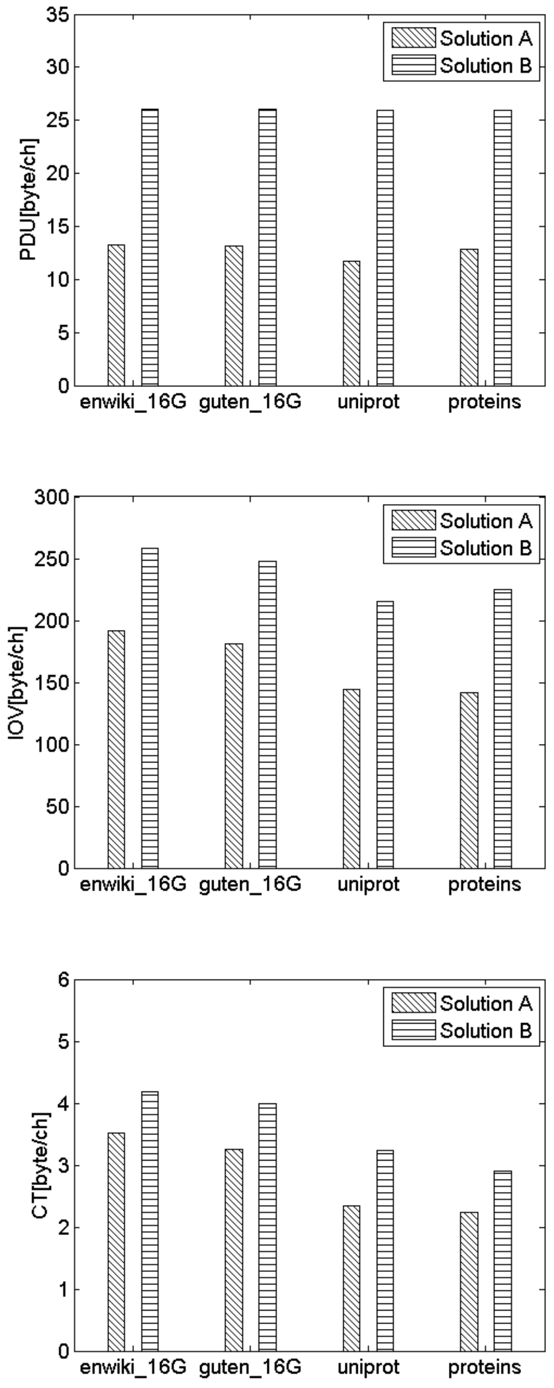

Fig. 4. A comparison of Solutions A and B on various corpora in terms of peak disk usage, I/O volume and construction time, where $D = 4$.