Efficient Suffix Array Construction and Checking in External Memory

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

Abstract—A suffix array (SA) must be checked to ensure its correctness. The existing SA checker verifies an SA by performing two passes of integer sorts that are time and space consuming especially when running in the external memory. Currently, building and checking an SA are done in sequence, where the time and the space bottlenecks are caused by building and checking the SA, respectively. In this paper, a fast and lightweight checking method is proposed to enable any induced sorting (IS) suffix sorting algorithm to verify a suffix array while it is being built. For performance evaluation, we first redesign the reduction phase of the external memory IS algorithm DSA-IS by employing the new substring sorting and naming methods to produce an optimized alternative DSA-IS+, and then apply the new checking method to DSA-IS+ for building and checking an SA simultaneously. Our experimental results indicate that, in comparison with that for building, the checking time and space complexities of eSAIS are substantial, but that of DSA-IS+ are negligible. As a result, DSA-IS+ can build and check an SA in a higher total time and space efficiencies.

Index Terms—suffix array, construction and verification, external memory.

1 Introduction

1.1 Background

The suffix array (SA) [1] is a data structure that has been widely used in many string processing applications, e.g., biological sequence alignment, time series analysis and text clustering. Given an input string, traversing its suffix tree can be emulated by using the corresponding enhanced suffix array [2], which mainly consists of the suffix and the longest common prefix arrays. It has been realized that the application scope of an index mainly depends on the construction speed and the space requirement. This leads to intensive works on designing time and space efficient suffix sorting algorithms over the past decade, assuming different computation models such as internal memory, external memory, parallel and distributed models. Particularly, to keep pace with the fast development of the data sampling techniques, several external memory algorithms have been proposed for building massive suffix arrays in recent years, e.g., DC3 [3], bwt-disk [4], SAscan [5], pSAscan [6], eSAIS [7], EM-SA-DS [8] and DSA-IS [9]. Among them, the latter three algorithms are based on the induced sorting (IS) method described in SA-IS [10].

The basic idea behind the induced sorting method is to induce the lexicographical order of all the substrings/suffixes from a sorted subset of substrings/suffixes. Following the idea, an IS-based suffix sorting algorithm is typically comprised of a reduction phase for sorting and naming substrings to reduce a string $x[0, n)$ to a short string $x1[0, n1)$ with $n1 \leq \frac{1}{2}n$ and an induction phase for sorting suffixes to induce $\mathsf{SA}(x)$ from $\mathsf{SA}(x1)$. During the two phases, the key operation is to retrieve the preceding character of a sorted substring/suffix. This can be done very quickly when x is fully accommodated in the internal memory, but will become slow when x resides in the external memory, as each operation takes a random disk access. For a high I/O efficiency, eSAIS, EM-SA-DS and DSA-IS use different auxiliary data structures to retrieve the preceding characters in a disk-friendly way. Particularly, both eSAIS and EM-SA-DS split a long substring into pieces and represent each piece by a fixed-size tuple, while DSA-IS does not. With an elaborate arrangement of the I/O operations, the programs for these three algorithms are competitive with those for others in terms of both time and space efficiencies.

1.2 Problems to Solve

Although the above three IS-based algorithms achieve great performance in comparison with prior arts, there still exist several drawbacks to be dealt with.

Firstly, the space requirement remains at a high level for all the three algorithms. In details, the peak disk usages for eSAIS, EM-SA-DS and DSA-IS are around $24n$, $31n$ and $20n$, respectively, the minimum of which is about three times as that for the lightweight suffix sorter SAscan (which I/O volume is super-linear). It is identified that DSA-IS reaches the peak disk usage when induced sorting substrings during the reduction phase. This reveals a need for designing new substring sorting and naming methods to improve the space efficiency in order to scale the problem size that can be tackled.

Secondly, a suffix array must be checked to ensure its correctness for later use. Currently, an SA built by each of these algorithms is verified using the SA checker described in [11], which performs two passes of external memory integer sorts that charge large disk space and long cpu time. As a result, the peak disk usage for both eSAIS and DSA-IS rises up to $26n$ while the time consumption also grows

- *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.*

by 20 percent as before, leading to a substantial performance degradation especially when processing large-scale datasets. Therefore, it is desired to reduce the check overhead in case it becomes a performance bottleneck.

1.3 Our Contribution

Our contribution in this paper mainly includes the following aspects.

- We redesign the reduction phase of DSA-IS by employing new methods for sorting and naming substrings. In our experiments, the program for the enhanced algorithm, called DSA-IS+, runs as fast as that for eSAIS while its peak disk usage is about two-thirds as that of eSAIS.
- We design a fast and lightweight SA checking method that can be seamlessly integrated into any IS-based suffix sorting algorithm. For performance evaluation, we combine this method with DSA-IS+ to produce a unified solution that can be employed to construct and check SA at the same time. It was observed that, by exploiting the use of our checking method, the performance overhead for verification is considerable small in comparison with that for construction. In details, the peak disk usage for the unified solution remains at the same level of that for DSA-IS+, while the growth in both time and I/O volume is negligible.

The rest of this paper is organized as follows. Section 2 gives the preliminaries in our presentation. Section 3 describes the substring sorting and naming methods for DSA-IS+, Section 4 the checking method and Section 5 the experimental results. Finally, we come to the conclusion in Section 6.

2 Preliminaries

2.1 Basic Notations

Given an input string $x[0,n) = x[0]x[1]...x[n-1]$ drawn from a totally ordered alphabet δ, we assume $x[n-1]$ to be the smallest character in δ that appears in x only once. For short, we use $\mathsf{sub}(x,i)$ and $\mathsf{suf}(x,i)$ to denote a substring running from $x[i]$ to the leftmost LMS character on its right side and a suffix running from $x[i]$ to $x[n-1]$, respectively. To follow lists the basic notations used in our presentation.

- L-type, S-type and LMS character/substring/suffix. We say $x[i]$ is S-type if (1) $i = n-1$ or (2) $x[i] < x[i+1]$ or (3) $x[i] = x[i+1]$ and $x[i+1]$ is S-type; otherwise, $x[i]$ is L-type. Furthermore, if $x[i]$ and $x[i-1]$ are respectively S-type and L-type, then $x[i]$ is also an LMS character. Moreover, if $x[i]$ is L-type, S-type or LMS, then $\mathsf{sub}(x,i)$ and $\mathsf{suf}(x,i)$ are L-type, S-type or LMS, respectively.
- preceding and succeeding character/substring/suffix. We say $x[i-1]$, $\mathsf{sub}(x,i-1)$ and $\mathsf{suf}(x,i-1)$ are the preceding character, substring and suffix for $x[i]$, $\mathsf{sub}(x,i)$ and $\mathsf{suf}(x,i)$, respectively. Similarly, we say $x[i+1]$, $\mathsf{sub}(x,i+1)$ and $\mathsf{suf}(x,i+1)$ are the succeeding character, substring and suffix for $x[i]$, $\mathsf{sub}(x,i)$ and $\mathsf{suf}(x,i)$, respectively.
- SA and STRA. We respectively use SA and STRA to denote the suffix and substring arrays. Specifically, both $\mathsf{SA}(x)$ and $\mathsf{STRA}(x)$ are permutations of $[0,n)$ such that $\mathsf{suf}(x,\mathsf{SA}(x)[i]) < \mathsf{suf}(x,\mathsf{SA}(x)[i+1])$ and $\mathsf{sub}(x,\mathsf{SA}(x)[i]) \le \mathsf{sub}(x,\mathsf{SA}(x)[i+1])$ in lexicographical order for all $i \in [0,n-1)$.
- BKT. Suffixes in $\mathsf{SA}(x)$ are naturally grouped into buckets according to their head characters. Given $\mathsf{BKT}(\mathsf{SA}(x),ch)$, the bucket occupies a consecutive range in $\mathsf{SA}(x)$ that gathers all the suffixes with a head character ch. Furthermore, a bucket, say $\mathsf{BKT}(\mathsf{SA}(x),ch)$, can be subdivided into $\mathsf{BKT_L}(\mathsf{SA}(x),ch)$ and $\mathsf{BKT_S}(\mathsf{SA}(x),ch)$ that only contain L-type and S-type suffixes in $\mathsf{BKT}(\mathsf{SA}(x),ch)$, respectively. Similarly, we define $\mathsf{BKT}(\mathsf{STRA}(x),ch)$, $\mathsf{BKT_L}(\mathsf{STRA}(x),ch)$ and $\mathsf{BKT_S}(\mathsf{STRA}(x),ch)$ for $\mathsf{STRA}(x)$.
- ISA. Inverse suffix array, satisfying $\mathsf{ISA}(x)[\mathsf{SA}(x)[i]] = i$ for all $i \in [0,n)$.
- PA. Position array, where $\mathsf{PA}(x)[i]$ records the position index of the i-th LMS substring from the left side of x.

2.2 An Overview of DSA-IS

As with other IS-based suffix sorting algorithms, DSA-IS mainly consists of a reduction phase followed by an induction phase. The induced sorting method is adopted in both two phases to sort substrings and suffixes, respectively.

2.2.1 Reduction Phase

Assume $x[0,n)$ and $\mathsf{STRA}(x)$ can be wholly accommodated into RAM, SA-IS performs the following three steps in the reduction phase to determine the lexicographical order of substrings.

step 1: Scan x rightward with i decreasing from $n-1$ to 0. For each scanned LMS character $x[i]$, insert i into the rightmost empty position of $\mathsf{BKT_S}(\mathsf{STRA}(x),ch)$.

step 2: Scan $\mathsf{STRA}(x)$ leftward with i increasing from 0 to $n-1$. For each scanned $p = \mathsf{STRA}(x)[i]$, if $ch = x[p-1]$ is the starting character of an L-type substring, then insert $p-1$ into the leftmost empty position of $\mathsf{BKT_L}(\mathsf{STRA}(x),ch)$.

step 3: Scan $\mathsf{STRA}(x)$ rightward with i decreasing from $n-1$ to 0. For each scanned $p = \mathsf{STRA}(x)[i]$, if $ch = x[p-1]$ is the starting character of an S-type substring, then insert $p-1$ into the rightmost empty position of $\mathsf{BKT_S}(\mathsf{STRA}(x),ch)$.

The key operation of steps 2-3 is to retrieve the preceding character $x[p-1]$ for currently scanned substring starting at $x[\mathsf{STRA}(x)[i]]$, where each operation takes a random access to the internal memory. A naive solution for extending the above 3-step procedure to external memory models is to replace each memory access with a disk access. However, this is not practical due to the large overhead for $\mathcal{O}(n)$ disk accesses. To reduce the I/O complexity, DSA-IS introduces the alternative $\mathsf{DSTRA}(x)$ for $\mathsf{STRA}(x)$ and employs I/O buffers to cache $\mathsf{DSTRA}(x)$ and other data structures for amortizing the disk access overhead, where the involved data structures are defined as below.

- DSTRA, $\mathsf{DSTRA_L}$ and $\mathsf{DSTRA_S}$. DSAITEM arrays. For an input x, each item of $\mathsf{DSTRA}(x)$ associates with a substring $\mathsf{sub}(x,p)$ and mainly consists of the following three components:
  - p: position index for $\mathsf{sub}(x,p)$.

– c: head character for $\mathsf{sub}(x, p)$, that is $x[p]$.
– t: type of the preceding character $x[p-1]$, set as 0 or 1 for L-type or S-type, respectively.

$\mathsf{DSTRA_L}(x)$ and $\mathsf{DSTRA_S}(x)$ only contain the items of L-type and S-type substrings in $\mathsf{DSTRA}(x)$, respectively.

- LMSDATA. A string array. Each item stores the characters of an LMS substring.
- LMSNAME. An integer array. Each item stores the name of an LMS substring, where the name represents the substring's lexicographical order among all.

Notice that the above data structures organize the items in the external memory according to the lexicographical order of their corresponding substrings.

Algorithm 1: The Reduction Phase for DSA-IS

Input: x
Output: $x1$
step 1: Partition x into blocks $b_1, b_2, ..., b_k$.
step 2: Induced sort the substrings of b_i for all $i \in [1, k]$ to compute $\mathsf{DSTRA_L}(b_i)$, $\mathsf{DSTRA_S}(b_i)$ and $\mathsf{LMSDATA}(b_i)$.
step 3: Induced sort the substrings of x from $\mathsf{DSTRA_L}(b_i)$, $\mathsf{DSTRA_S}(b_i)$ and $\mathsf{LMSDATA}(b_i)$ to compute $\mathsf{DSTRA}(x)$ and $\mathsf{LMSDATA}(x)$. Meanwhile, scan $\mathsf{DSTRA}(x)$ to compute $\mathsf{DSTRA_{LMS}}(x)$.
step 4: Scan $\mathsf{DSTRA_{LMS}}(x)$ and $\mathsf{LMSDATA}(x)$ to name the sorted LMS substrings for producing $\mathsf{LMSNAME}(x)$.
step 5: Compute the reduced string $x1$ from $\mathsf{LMSNAME}(x)$ and $\mathsf{DSTRA_{LMS}}(x)$.

The algorithmic framework of the reduction phase for DSA-IS is shown in Algorithm 1, where each step is sketched below.

Step 1 scans x leftward to sequentially retrieve the LMS substrings and inserts them one by one into the blocks with a capacity $m = \mathcal{O}(\mathsf{M})$, where M is the size of the internal memory. In this way, each block is either a single-block composed of one LMS substring or a multi-block composed of at least two successive LMS substrings. Particularly, each multi-block contains no more than m characters such that it can be processed in the internal memory during the two phases, while a single-block may not be so.

Step 2 adopts different strategies to tackle b_i with respect to whether the block is a single-block or a multi-block. Specifically, if b_i is a multi-block, then it computes $\mathsf{STRA}(b_i)$ by conducting the substring sorting algorithm of SA-IS and scans $\mathsf{STRA}(b_i)$ and b_i to obtain $\mathsf{DSTRA_L}(b_i)$, $\mathsf{DSTRA_S}(b_i)$ and $\mathsf{LMSDATA}(b_i)$; otherwise, the substrings in the block are already sorted and it directly scans b_i to obtain the three arrays.

Step 3 also computes $\mathsf{DSTRA}(x)$ following the induced sorting method. In details, when inducing substrings in $\mathsf{DSTRA}(x)$, it retrieves the head character of the L-type/S-type preceding substring for the currently scanned substring from the external memory by using sequential I/O operations. This is feasible because the corresponding DSAITEM of the preceding substring is located at the leftmost unvisited position in $\mathsf{DSTRA_L}(b_i)/\mathsf{DSTRA_S}(b_i)$, where the subscript i of the block that contains the target item can be determined in $\mathcal{O}(1)$ time using $\mathcal{O}(\frac{n}{m})$ space [9].

Step 4 names the sorted LMS substrings according to their lexicographical order. For each pair of the neighboring substrings in $\mathsf{DSTRA_{LMS}}(x)$, it conducts a string comparison to check if they are equal, where their characters can be sequentially retrieved from $\mathsf{LMSDATA}(x)$. If yes, then the two substrings have a same name; otherwise, the name for the latter is greater than the former by one.

Step 5 sorts the names in $\mathsf{LMSNAME}(x)$ by $\mathsf{DSTRA_{LMS}}(x)[i].p$ for all $i \in [0, n1)$ to produce the reduced string $x1$.

2.2.2 Induction Phase

Algorithm 2: The Induction Phase for DSA-IS

Input: x, $\mathsf{PA}(x)$, $\mathsf{DSA}(x1)$
Output: $\mathsf{DSA}(x)$
Step 1: Compute $\mathsf{DSA_{LMS}}(x)$ from $\mathsf{DSA}(x1)$ and decompose $\mathsf{DSA_{LMS}}(x)$ into $\mathsf{DSA_{LMS}}(b_i)$ for all $i \in [1, k]$.
Step 2: Induced sort the suffixes of b_i from $\mathsf{DSA_{LMS}}(b_i)$ to compute $\mathsf{DSA_L}(b_i)$ and $\mathsf{DSA_S}(b_i)$ for all $i \in [1, k]$.
Step 3: Induced sort the suffixes of x from $\mathsf{DSA_L}(b_i)$ and $\mathsf{DSA_S}(b_i)$ to compute $\mathsf{DSA}(x)$.

As shown in Algorithm 2, the induction phase of DSA-IS consists of three steps, where the data structures for the induction phase are similar to their counterparts for the reduction phase:

- $\mathsf{DSA_{LMS}}$. A DSAITEM array. For an input x, each item of $\mathsf{DSA_{LMS}}(x)$ is associated with an LMS suffix of x.
- DSA, $\mathsf{DSA_L}$ and $\mathsf{DSA_S}$. DSAITEM arrays. For an input x, each item of $\mathsf{DSA}(x)$ is associated with a suffix of x. $\mathsf{DSA_L}(x)$ and $\mathsf{DSA_S}(x)$ only contain the items of L-type and S-type suffixes in $\mathsf{DSA}(x)$, respectively.

The first step in Algorithm 2 is to induce the order of LMS substrings of x. For the purpose, it sorts the items of $\mathsf{DSA}(x1)$ by $\mathsf{DSA}(x1)[i].p$ for all $i \in [0, n1)$ to compute $\mathsf{ISA}(x1)$ and then sorts the items of $\mathsf{PA}(x)$ by $\mathsf{ISA}(x1)[j]$ for all $j \in [0, n1)$ to compute $\mathsf{DSA_{LMS}}(x)$. Afterward, it decomposes $\mathsf{DSA_{LMS}}(x)$ into $\mathsf{DSA_{LMS}}(b_i)$ for all $i \in [1, k]$ and reuses steps 2-3 of the reduction phase to compute $\mathsf{DSA}(x)$ by replacing $\mathsf{DSTRA_L}(b_i)$ and $\mathsf{DSTRA_S}(b_i)$ with $\mathsf{DSA_L}(b_i)$ and $\mathsf{DSA_S}(b_i)$, respectively.

3 Details of DSA-IS+

It was observed from our experiments that, the program for DSA-IS outperforms that for eSAIS with respect to the space efficiency, but it runs slower than the latter due to the large I/O volume for sorting and naming substrings. In this section, we present two methods that can substantially reduce the construction time and I/O volume of the reduction phase without a sacrifice in its high space efficiency. We demonstrate in Section 5 that, by exploiting the use of these methods, the optimized version of DSA-IS, namely DSA-IS+, runs as fast as eSAIS and its peak disk usage is two-thirds as that of the latter.

3.1 Method A

Recall that in Algorithm 1, DSA-IS first sorts LMS substrings in steps 2-3 and then names the substrings in their sorted order in step 4. These two procedures can be merged by making use of the following fact: $\forall i, j, \in [0, n)$, $\mathsf{sub}(x, i)$ and $\mathsf{sub}(x, j)$ are equal if and only if $x[i] = x[j]$ and $\mathsf{sub}(x, i + 1) = \mathsf{sub}(x, j + 1)$. Specifically, for any two neighboring substrings in $\mathsf{DSTR}(x)$, say $\mathsf{DSTR}(x)[p]$ and $\mathsf{DSTR}(x)[p+1]$, there must have $\mathsf{DSTR}(x)[p] \leq \mathsf{DSTR}(x)[p + 1]$. Hence, the only information required for naming the two substrings is to check if they are equal or not. Following the fact describe above, this can be trivially done by adding two fields $r1$ and $r2$ to each item of $\mathsf{DSTR}(x)$, where $r1$ and $r2$ respectively record the names of the corresponding substring and the succeeding substring. More specifically, assume all the items of LMS characters are already inserted into the corresponding buckets of $\mathsf{DSTR}(x)$, it takes the following two steps to sort and name substrings at the same time:

step 1: Initially set $r = 0$. Scan $\mathsf{DSTR}(x)$ rightward. For each scanned item $e1$ and previously scanned item $e2$, if $e1.r2 = e2.r2$ and $e1.ch = e2.ch$ then set $e1.r1 = e2.r1$; otherwise, set $e1.r1 = r$ and increase r by one. If $e1.p > 0$ and $e1.t = 0$, then determine the block b_i that contains $\mathsf{sub}(x, e1.p - 1)$, retrieve the leftmost unvisited item $e3$ from $\mathsf{DSTR_L}(b_i)$, set $e3.r2 = e1.r1$ and insert $e3$ into the leftmost empty position of $\mathsf{BKT_L}(\mathsf{DSTR}(x), e3.c)$.

step 2: Initially set $r = n-1$. Scan $\mathsf{DSTR}(x)$ leftward. For each scanned item $e1$ and previously scanned item $e2$, if $e1.r2 = e2.r2$ and $e1.ch = e2.ch$ then set $e1.r1 = e2.r1$; otherwise, set $e1.r1 = r$ and decrease r by one. If $e1.p > 0$ and $e1.t = 1$, then determine the block b_i that contains $\mathsf{sub}(x, e1.p - 1)$, retrieve the leftmost unvisited item $e3$ from $\mathsf{DSTR_S}(b_i)$, set $e3.r2 = e1.r1$ and insert $e3$ into the rightmost empty position of $\mathsf{BKT_S}(\mathsf{DSTR}(x), e3.c)$.

At the end of step 2, all the LMS substrings are already sorted and their names can be obtained by copying $r1$ from the items of $\mathsf{DSTR}(x)$.

3.2 Method B

A solution to reducing the I/O volume is to employ a multi-way merge algorithm for combining sorted LMS substrings of all the blocks, where the task of the merger is to sort and name LMS substrings by literally comparing their characters. A string comparison can be done very quickly if the involved two substrings are wholly loaded into RAM. However, there may exist some LMS substrings that cannot be accommodated in the internal memory. For these long substrings, the method described in Section 3.1 is employed. Following the ideas, we revise steps 2-4 of Algorithm 1 as below, where $i \in [1, k]$.

step 2′: Induced sort the substrings of b_i to compute $\mathsf{LSA_L}(b_i)$, $\mathsf{LSA_S}(b_i)$, $\mathsf{DSTRA'_L}(b_i)$ and $\mathsf{DSTRA'_S}(b_i)$.

step 3′: Induced sort the substrings embraced in the long LMS substrings of x from $\mathsf{LSA_L}(b_i)$, $\mathsf{DSTRA'_L}(b_i)$ and $\mathsf{DSTRA'_S}(b_i)$ to compute $\mathsf{LSA_L}(x)$.

step 4′: Merge $\mathsf{LSA_L}(x)$ and $\mathsf{LSA_S}(b_i)$ by using a min heap to compute $\mathsf{LMSNAME}(x)$.

The definitions of the newly introduced notations are given below.

- long/short LMS substring and threshold value D. All the LMS substrings are classified into two categories: long and short. Specifically, an LMS substring is short if it contains no more than D characters; otherwise, it is long.
- $\mathsf{DSTR'}$, $\mathsf{DSTRA'_L}$ and $\mathsf{DSTRA'_S}$. DSAITEM arrays. For an input x, $\mathsf{DSTR'}(x)$ only contains the items in $\mathsf{DSTR}(x)$ that associate with substrings embraced in long LMS substrings. We say $\mathsf{sub}(x, p)$ embraces $\mathsf{sub}(x, q)$ if the two substrings end with the same LMS character and $p \leq q$. $\mathsf{DSTRA'_L}(x)$ and $\mathsf{DSTRA'_S}(x)$ respectively contain the items in $\mathsf{DSTRA'}(x)$ that associate with L-type and S-type substrings.
- $\mathsf{LSA_L}$ and $\mathsf{LSA_S}$. LSAITEM arrays. For an input x, each item of $\mathsf{LSA_L}(x)/\mathsf{LSA_S}(x)$ associates with a long/short LMS substring $\mathsf{sub}(x, p)$ and mainly consists of four components:
  - p: position index for $\mathsf{sub}(x, p)$.
  - s: leftmost D characters of $\mathsf{sub}(x, p)$, if it is long; otherwise, all the characters.
  - t: type of the last character in s, set as 0 or 1 for L-type or S-type, respectively.
  - r: name of $\mathsf{sub}(x, p)$.

We describe each step detailedly in what follows.

Step 2′

This step consists of two substeps:

(a) Compute $\mathsf{DSTRA'_L}(b_i)$ and $\mathsf{DSTRA'_S}(b_i)$ by reusing the algorithm for computing $\mathsf{DSTRA_L}(b_i)$ and $\mathsf{DSTRA_S}(b_i)$ in step 2 of Algorithm 1.

(b) Scan $\mathsf{DSTRA'_S}(b_i)$ rightward. For each scanned item $e1$ with $e1.t = 0$, create an LSAITEM $e2$ for the current LMS substring, compute $e2.t$ and $e2.s$ by visiting b_i, and insert $e2$ into $\mathsf{LSA_L}(b_i)$ or $\mathsf{LSA_S}(b_i)$ according to its length.

Step 3′

This step consists of three substeps:

(a) Scan x leftward with i decreasing from $n - 1$ to 0. For each scanned character $x[i]$, if $x[i]$ is an ending character of a long LMS substring, insert $\mathsf{sub}(x, i)$ into the rightmost empty position in $\mathsf{BKT_S}(\mathsf{DSTRA'}(x), x[i])$.

(b) Reuse step 1 of method A to sort and name the L-type substrings embraced in long LMS substrings of x by replacing $\mathsf{DSTR}(x)$ with $\mathsf{DSTR'}(x)$.

(c) Reuse step 2 of method A to sort and name the S-type substrings embraced in long LMS substrings of x by replacing $\mathsf{DSTR}(x)$ with $\mathsf{DSTR'}(x)$. For each scanned item $e1$ that associates with an LMS substring, determine the block b_i that contains $\mathsf{sub}(x, e1.p)$, retrieve the leftmost unvisited item $e4$ from $\mathsf{LSA_L}(b_i)$, set $e4.r = e1.r1$ and insert $e4$ to the front of $\mathsf{LSA_L}(x)$.

Step 4′

Initialize the heap by inserting the leftmost item of each sorted sequence of $\mathsf{LSA_L}(x)$ and $\{\mathsf{LSA_S}(b_1), ..., \mathsf{LSA_S}(b_k)\}$. Sequentially pop the items in the heap. For currently popped item $e1$, insert into the heap the next item $e2$ of the same sequence and compare $e1$ with the previously popped item $e3$ to determine the names of their corresponding LMS

substrings. In this step, each string comparison for heap-sorting items and naming substrings observes the following rules, where $e1$ and $e2$ are items to be compared.

- rule 1: If both items belong to $\{\mathsf{LSA_S}(b_1), ..., \mathsf{LSA_S}(b_k)\}$, then literally compare $e1.s$ with $e2.s$ to determine their order.
- rule 2: If both items belong to $\mathsf{LSA_L}(x)$, then directly compare $e1.r$ with $e2.r$ to determine their order.
- rule 3: If one belongs to $\mathsf{LSA_L}(x)$ and the other belongs to $\{\mathsf{LSA_S}(b_1), ..., \mathsf{LSA_S}(b_k)\}$, then literally compare $e1.s$ with $e2.s$. If equal, then continue to compare $e1.t$ with $e2.t$.

Clearly, the time overhead of each string comparison is upper-bounded by the threshold value D.

4 LIGHTWEIGHT SA CHECKER

It is essential for a suffix sorting algorithm to check the correctness of the output suffix arrays. Several existing SA construction algorithms verify their outputs based on Lemma 4.1 [11].

Lemma 4.1. An array $\mathsf{SA}[0, n)$ is the suffix array of a string $x[0, n)$ if and only if the following conditions are satisfied for all $i, j \in [0, n)$ and $i \neq j$:
(1) $\mathsf{SA}[i], \mathsf{SA}[j] \in [0, n)$ and $\mathsf{SA}[i] \neq \mathsf{SA}[j]$.
(2) $\mathsf{ISA}[i] < \mathsf{ISA}[j] \Leftrightarrow (x[i], \mathsf{ISA}[i+1]) < (x[j], \mathsf{ISA}[j+1])$.

This SA checker performs two passes of integer sorts, where the first pass checks if $\mathsf{SA}[0, n)$ is a permutation of $[0, n)$ for condition (1) and the second pass checks if the order of suffixes in each SA bucket is correct for condition (2). Although this method runs fast in the internal memory, the required disk space and I/O volume are considerably huge when it sorts items using the external memory, resulting in a non-negligible performance overhead. Take eSAIS for example, the peak disk usages for constructing and checking SA are around 24n and 27n, respectively, while the overall I/O volume also increases by 50n for conducting the two integer sorts.

4.1 Details

We describe a new SA checking method that can be seamlessly integrated into any IS-based induced sorting algorithm, which has almost no influence on the overall performance. The main idea is to verify the two conditions listed in Lemma 4.1 during the induction phase. In what follows, we first describe two observations that are always true in the process of induced sorting suffixes, where $\mathsf{SA_{LMS1}}(x)$ is the position index array for the sorted LMS suffixes produced by sorting $\mathsf{PA}(x)$ with $\mathsf{ISA}(x1)$ at the beginning of the induction phase, and $\mathsf{SA_{LMS1}}(x)[u]$ and $\mathsf{SA_{LMS1}}(x)[v]$ are the ending position indexes of the LMS substrings that embrace $x[\mathsf{SA}(x)[i]]$ and $x[\mathsf{SA}(x)[j]]$, respectively.

Observation 4.2. $\forall i, j \in [0, n)$, $\mathsf{SA}(x)[i] = \mathsf{SA}(x)[j]$ *and* $i \neq j \Leftrightarrow \mathsf{SA_{LMS1}}(x)[u] = \mathsf{SA_{LMS1}}(x)[v]$ *and* $u \neq v$.

Proof: Omit. □

Observation 4.3. $\forall i, j \in [0, n)$, $i < j$ *and* $\mathsf{suf}(x, \mathsf{SA}(x)[i]) > \mathsf{suf}(x, \mathsf{SA}(x)[j]) \Rightarrow u < v$ *and* $\mathsf{suf}(x, \mathsf{SA_{LMS1}}(x)[u]) > \mathsf{suf}(x, \mathsf{SA_{LMS1}}(x)[v])$.

Proof: An IS-based SA construction algorithm induces the order of suffixes by utilizing the fact that $\mathsf{suf}(x, p_1) < \mathsf{suf}(x, q_1) \Leftrightarrow (x[p_1], \mathsf{suf}(x, p_1 + 1)) < (x[q_1], \mathsf{suf}(x, q_1 + 1)$. The algorithm guarantees that $\mathsf{ISA}(x)[p_1] < \mathsf{ISA}(x)[q_1] \Leftrightarrow (x[p_1], \mathsf{ISA}(x)[p_1 + 1]) < (x[q_1], \mathsf{ISA}(x)[q_1 + 1])$. This indicates that $\mathsf{suf}(x, p_1) < \mathsf{suf}(x, q_1)$ and $\mathsf{ISA}(x)[p_1] > \mathsf{ISA}(x)[q_1]$ only if $\mathsf{suf}(x, p_2) < \mathsf{suf}(x, q_2)$ and $\mathsf{ISA_{LMS1}}(x)[p_2] > \mathsf{ISA_{LMS1}}(x)[q_2]$, where p_2 and q_2 are the position indexes of the leftmost LMS characters on the right side of $x[p_1]$ and $x[q_1]$, respectively. □

The above observations are used to prove the correctness of the following statement.

Theorem 4.4. For any IS-based induced sorting algorithm, the output array $\mathsf{SA}(x)$ is the suffix array of the input string $x[0, n)$ if and only if the following conditions are satisfied:
(1) $\forall i, j \in [0, n)$ and $i \neq j$, $\mathsf{SA_{LMS1}}(x)[i] \neq \mathsf{SA_{LMS1}}(x)[j]$.
(2) $\mathsf{SA_{LMS1}(x)} = \mathsf{SA_{LMS2}(x)}$, where $\mathsf{SA_{LMS2}}$ is the position index array for the sorted LMS suffixes retrieved from $\mathsf{SA}(x)$ at the end of the induction phase.

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

Suppose $\mathsf{SA}(x)$ is not a permutation of $[0, n)$. Let $i \neq j$ and $\mathsf{SA}(x)[i] = \mathsf{SA}(x)[j]$, then there must exist u, v satisfying $u \neq v$ and $\mathsf{SA_{LMS1}}(x)[u] = \mathsf{SA_{LMS1}}(x)[v]$ according to Observation 4.2. This violates the first condition in Theorem 4.4.

Suppose $\mathsf{suf}(x, i) > \mathsf{suf}(x, j)$ and $\mathsf{ISA}(x)[i] < \mathsf{ISA}(x)[j]$. Then, there must exist u, v satisfying $\mathsf{suf}(x, u) > \mathsf{suf}(x, v)$ and $\mathsf{ISA_{LMS1}}(x)[u] < \mathsf{ISA_{LMS1}}(x)[v]$ according to Observation 4.3. Let (u_0, v_0) be the pair that maximize $u + v$, then we have $\mathsf{ISA_{LMS2}}(x)[u_0] > \mathsf{ISA_{LMS2}}(x)[v_0]$. This violates the second condition in Theorem 4.4. □

To verify the conditions in Theorem 4.4, we can design a checker based on the following idea. The first condition can be checked after sorting $\mathsf{PA}(x)$ by $\mathsf{ISA}(x1)$ at the beginning of the induction phase. For the second condition, a naive method that takes $\mathcal{O}(n)$ time and space is to copy $\mathsf{SA_{LMS1}}(x)$ and $\mathsf{SA_{LMS2}}(x)$ and then compare their items in sequence at the end of the induction phase. An alternative for lowering the overhead is to compute the fingerprints for $\mathsf{SA_{LMS1}}(x)$ and $\mathsf{SA_{LMS2}}(x)$ by using fingerprint functions and then compare the fingerprints instead of the items in these arrays to determine their equality. Obviously, $\mathsf{SA_{LMS1}}(x)$ and $\mathsf{SA_{LMS2}}(x)$ have a common fingerprint if they are identical, but the inverse is not always true. Fortunately, the probability of a false match can be reduced to a completely negligible level by properly setting the parameters for the calculation formula. This leads us to Theorem 4.5.

Theorem 4.5. For any IS-based induced sorting algorithm, the output array $\mathsf{SA}(x)$ is the suffix array of the input string $x[0, n)$ with a high probability if the following conditions are satisfied:
(1) $\forall i, j$, $\mathsf{SA_{LMS1}}(x)[i] \neq \mathsf{SA_{LMS1}}(x)[j]$.
(2) $\mathsf{FP(SA_{LMS1}(x))} = \mathsf{FP(SA_{LMS2}(x))}$.

4.2 Implementation

We choose the approach presented in [12] to compute the fingerprints in need. Specifically, for a given array $x[0, n)$,

the fingerprint $\mathsf{FP}(x)[0,n)$ can be iteratively computed by the formula $\mathsf{FP}(x)[i,j] = \sum_{p=i}^{j} \alpha^{j-p} \cdot x[p] \, mod \, \beta$, where β is a large prime and α is an integer randomly chosen from domain $[1,\beta)$. In this way, we can slightly adapt Algorithm 2 to compute $\mathsf{FP}(\mathsf{SA}_{\mathsf{LMS1}}(\mathsf{x}))$ by scanning $\mathsf{DSA}_{\mathsf{LMS}}(x)$ in step 1 and $\mathsf{FP}(\mathsf{SA}_{\mathsf{LMS2}}(\mathsf{x}))$ by scanning $\mathsf{DSA}(x)$ during the time when inducing S-type suffixes in step 3, respectively. Obviously, the time, space and I/O complexities of the implementation are linearly related to n.

5 Experiments

For performance evaluation, we engineer DSA-IS and its optimized DSA-IS+, by using the STXXL's containers (sorter, priority queue, vector and stream) to conduct efficient I/O operations in the external memory. As reported in [5], SAscan outperforms eSAIS in practice when the internal memory capacity M is adequate. However, the time and space complexities of the program for SAscan are $\widetilde{\mathcal{O}}(n^2/M)$ and $\widetilde{\Omega}(n^2/M)$, respectively, thus it will suffer from a performance degradation as the size of the input string gets larger.[1] On the other hand, the programs for DSA-IS, DSA-IS+ and eSAIS[2] are not only of linear I/O volume and space complexities in theory, but also commonly implemented by the STXXL library. **Therefore, we use eSAIS rather than SAscan as a baseline for evaluating the performance of DSA-IS and DSA-IS+.** The experimental platform is a desktop computer equipped with an Intel Xeon E3-1220 V2 CPU, 4GiB RAM and 500GiB HD. 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 of the programs.

- construction time (CT): the running time, in units of microseconds per character.
- peak disk usage (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.

5.1 Performance Evaluation on Construction Algorithms

Figure 1 illustrates the performance of the programs for DSA-IS and eSAIS in terms of the investigated three metrics, where guten_14G and enwiki_14G are prefixes of the corresponding corpus. As depicted, the peak disk usage for DSA-IS is around $16n$ on the two datasets, which is two-thirds of that for eSAIS. However, eSAIS outperforms DSA-IS with respect to the construction time, where the speed gap between them is mainly due to the different I/O volumes taken by the two algorithms. In order for a deep insight, we show in Table 2 the statistics for I/O volumes of the algorithms in their reduction and induction phases. It can be observed that, although the I/O volumes spent in the induction phase are similar, eSAIS is far more I/O-efficient than DSA-IS when sorting substrings during the reduction phase. More specifically, the ratio of reduction and induction volumes for DSA-IS is about 0.80 while the counterpart for eSAIS is 0.23.

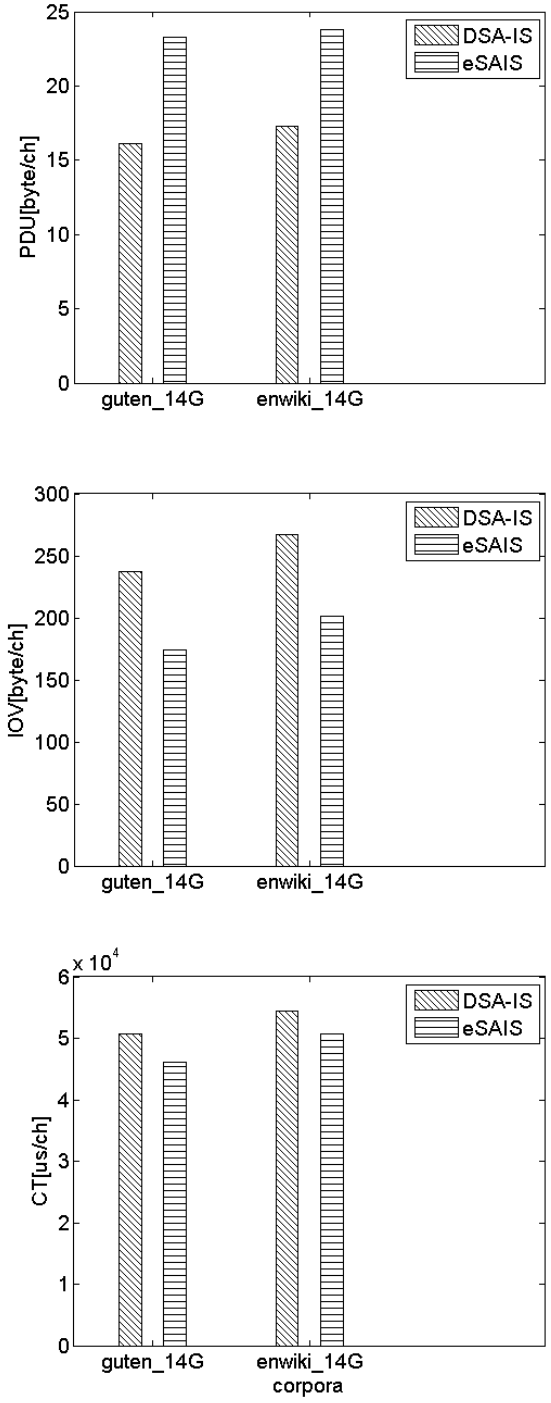


Fig. 1. Experimental results for DSA-IS and eSAIS on guten_14G and enwiki_14G in terms of peak disk usage, I/O volume and construction time.

The above phenomenon can be explained as follows. During the reduction phase, DSA-IS partitions the input

1. The authors in [6] says, "SAscan needs just 7.5n bytes of disk space but because of its $\widetilde{\mathcal{O}}(n^2/M)$ time complexity, it is competitive with eSAIS only when the input is less than about five times the size of RAM.".

2. https://panthema.net/2012/1119-eSAIS-Inducing-Suffix-and-LCP-Arrays-in-External-Memory/

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.
genome	2.9	6	Human genome data, used in Dementiev et al. [3], at http://algo2.iti.kit.edu/dementiev/esuffix/instsances.shtml.

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

	eSAIS				DSA-IS				DSA-IS+ ($D_1 = 8$, $D_2 = 10$)			
Size	Red.	Ind.	Total	Ratio	Red.	Ind.	Total	Ratio	Red.	Ind.	Total	Ratio
1G	36.6	132.8	169.4	0.27	81.3	105.9	187.2	0.76	39.1	104.3	143.4	0.37
2G	36.0	141.9	177.9	0.25	83.5	108.2	191.7	0.77	37.8	106.4	144.2	0.35
4G	35.6	152.1	187.7	0.23	94.3	118.9	213.2	0.79	42.4	117.0	159.4	0.36
8G	35.2	165.7	200.9	0.21	107.8	132.3	240.1	0.81	43.7	129.9	173.6	0.33
14G	35.0	172.1	207.1	0.20	121.9	146.6	268.5	0.83	43.8	142.3	186.1	0.30

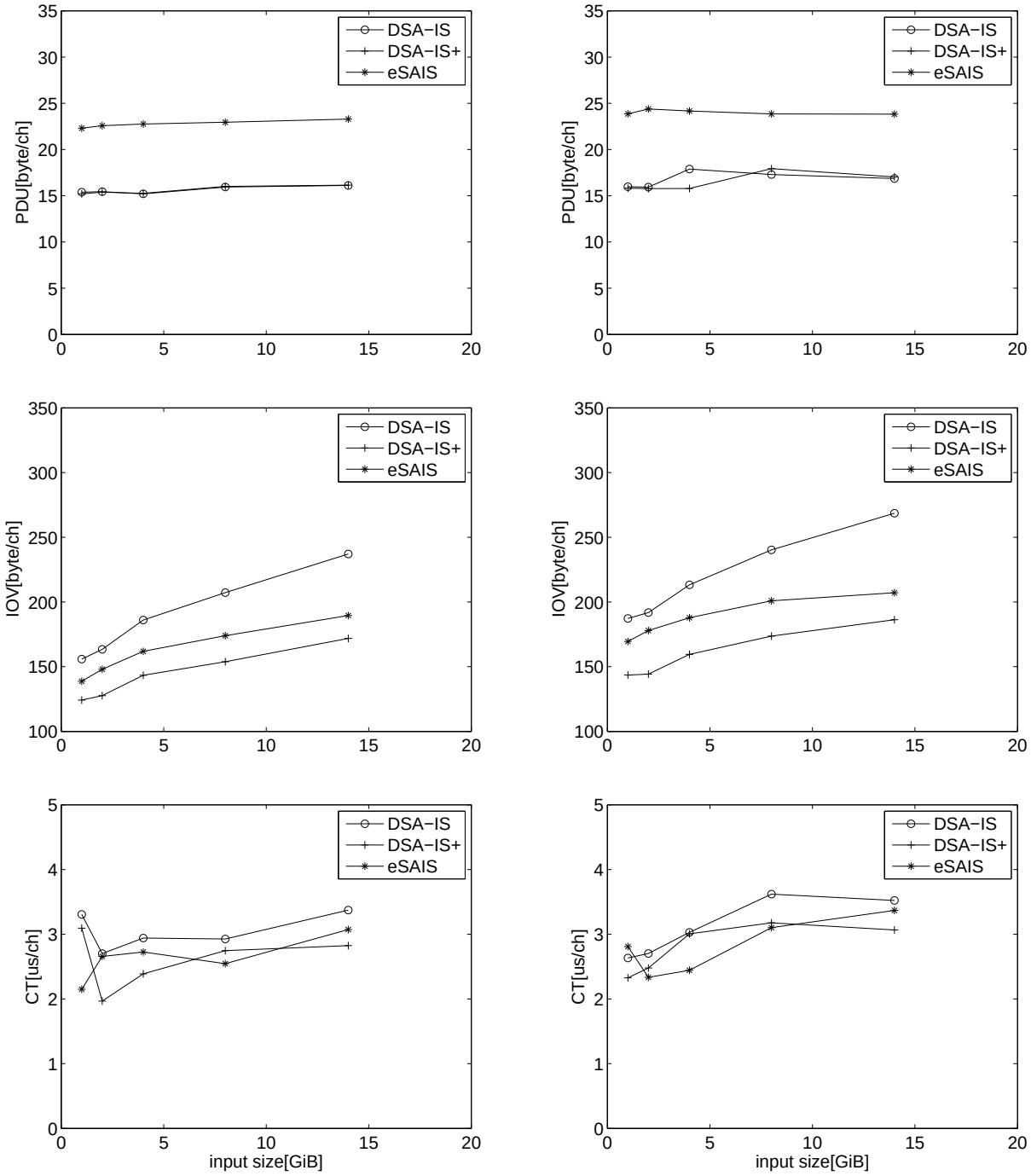
string x into multiple blocks and induced sorts the substrings of each block in the internal memory. Then it reuses the induced sorting method to sort all the substrings of x in the external memory by merging the block-wise results residing on disks. In contrast with DSA-IS, eSAIS splits x into fixed-size partitions and sorts the substrings of each partition in the internal memory by calling std::sort. Then, it generates the global result by merging the partial ones with an internal memory heap. This leads to a higher I/O efficiency against DSA-IS.

We attempt to reduce the I/O overhead for DSA-IS by using the substring sorting and naming methods described in Section 3. Our program for the enhanced algorithm DSA-IS+ introduces two parameters D_1 and D_2 to respectively specify the threshold value D for classifying the substrings in x and the reduced strings $\{x1, x2, ...\}$ according to the instructions in Section 3.2. As observed from Table 2, the total I/O volumes for DSA-IS+ and eSAIS are similar and the ratio of the reduction and induction I/O volumes for either of them is no more than half of that for DSA-IS. This substantial improvement leads to a great speedup against DSA-IS. Figure 2 shows the performance trends of DSA-IS+ and eSAIS on guten and enwiki as the corpora size increases from $1G$ to $14G$, which indicates that the speed of DSA-IS+ is similar to that of eSAIS and its peak disk usage remains at a low level as the same as that of DSA-IS.

A series of experiments are conducted to investigate the effect of D_1 and D_2 on the performance of DSA-IS+. As shown in Figure 3, the time consumption of DSA-IS+ gets smaller as D_2 becomes larger, while the peak disk usage and I/O volume are almost unchanged. Furthermore, the data of Table 3 indicates that a small variation on the values of D_1 and D_2 can incur a significant fluctuation on the speed of DSA-IS+. For instance, when changing (D_1, D_2) from $(8, 10)$ to $(16, 16)$, the construction time for the uniprot dataset decreases from 2.64 to 2.24 microseconds per character. This is because the proportions of long LMS substrings of x and $\{x1, x2, ...\}$ decrease when D_1 and D_2 increase. Recall that, after partitioning x into blocks, DSA-IS+ sorts the long LMS substrings of each block in RAM and merges them by an external memory sorter, which is time-consuming when the number of long LMS substrings is large. In our program, the ratio of long and short LMS substrings can be controlled by adjusting the values of D_1 and D_2. However, when D_1 and D_2 get larger, the min heap also takes more time to merge the sorted short and long LMS substrings, as each string comparison takes $\mathcal{O}(D_1)$ and $\mathcal{O}(D_2)$ time in the first and recursion levels, respectively. Fortunately, Table 4 shows that the majority of LMS substrings in real datasets are considerably short. In practice, we can set D_1 and D_2 according to the statistics of the length distribution of LMS substrings, where the statistics can be collected when partitioning x at the beginning of the reduction phase.

5.2 Performance Evaluation on Checking Methods

We make a performance comparison of the proposed checking method and the work presented in [11], which are denoted by CM1 and CM2, respectively. For performance evaluation, we integrate CM1 into the program of DSA-IS+ following Section 4.2 and reuse the implementation for CM2 in the program of eSAIS to verify the outputs of DSA-IS+ and eSAIS. Table 5 gives a glimpse of the performance overhead for the two checking methods. It can be seen that, when checking the suffix array of enwiki_8G by using CM2, both time and I/O volume for verification are about one-fifth of that for construction. On the other hand, CM1 has almost no negative impact on the overall performance. As demonstrated in lines 3 and 5, the peak disk usage for CM1 is no more than that for DSA-IS+ and the increase in the I/O volume can be ignored. It worth mentioning that the data of the running time for the combination of DSA-IS+ and CM1 is a bit shorter than that for the plain DSA-IS+.



(a) guten (b) enwiki

Fig. 2. Experimental results for DSA-IS, DSA-IS+ and eSAIS on guten and enwiki in terms of peak disk usage, I/O volume and construction time, where $D_1 = 8$, $D_2 = 10$ and the input size varies in $\{1, 2, 4, 8, 14\}$ GiB.

TABLE 3
Effects of D_1 and D_2 for DSA-IS+

	eSAIS			DSA-IS+									
Corpora	PDU	IOV	CT	D_1	D_2	PDU	IOV	CT	D_1	D_2	PDU	IOV	CT
uniprot	22.71	162.01	2.50	8	10	14.27	146.10	2.64	16	16	14.27	144.90	2.24
proteins	24.09	172.29	2.33	8	10	16.12	147.92	2.28	20	12	16.12	147.85	2.14
genome	22.64	157.41	2.15	8	10	15.34	142.05	2.75	20	12	15.34	140.89	2.25

TABLE 4
Statistics on the Length Distribution of Long and Short LMS Substrings for DSA-IS+

level 0										
Corpora	D_1	long	short	total	ratio	D_1	long	short	total	ratio
uniprot	8	30243605	785997413	816241018	0.963	16	3850755	812390263	816241018	0.999
proteins	8	1687673	377404329	379092002	0.999	20	10930	379081072	379092002	0.999
genome	8	19446841	773670266	793117107	0.975	20	509998	792607109	793117107	0.999
level 1										
Corpora	D_2	long	short	total	ratio	D_2	long	short	total	ratio
uniprot	10	11849	263748775	263760624	0.995	16	380	263760244	263760624	0.999
proteins	10	18126	123772731	123790857	0.996	20	2937	123787920	123790857	0.999
genome	10	311607	248243216	248554823	0.999	20	153845	248400978	248554823	0.999

TABLE 5
Overall Performance of DSA-IS+ and eSAIS with CM1 and CM2 on enwiki_8G

	DSA-IS+ ($D_1 = 8$, $D_2 = 10$)			eSAIS		
Checking Method	PDU	IOV	CT	PDU	IOV	CT
CM1	17.93	180.01	3.10	-	-	-
CM2	26.00	231.68	4.00	27.00	246.93	3.74
no check	17.93	173.67	3.17	23.85	200.93	3.10

This is a normal phenomenon due to the fluctuating I/O performance, which in turn indicates that the time overhead consumed by CM1 is also negligible.

6 Conclusion

For better performance, we redesign the reduction phase of DSA-IS by employing two methods for sorting and naming substrings. The program of the enhanced algorithm DSA-IS+ is engineered by the STXXL library to achieve a high I/O efficiency. Our experiments indicate that DSA-IS+ requires only around $16/24 = 0.67$ peak disk usage as that for eSAIS and runs as fast as the latter on various real datasets.

We describe an SA checker of linear time and space complexities that can be used to verify a suffix array during the time when it is being built. In combination with DSA-IS+, our checking method almost has no influence on the overall performance and is more efficient than the existing work in terms of time, space and I/O efficiencies.

References

[1] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-line String Searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[2] M. Abouelhodaa, S. Kurtzb, and E. Ohlebuscha, "Replacing Suffix Trees with Enhanced Suffix Arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, November 2004.

[3] 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.

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

[5] 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.

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

[7] T. Bingmann, J. Fischer, and V. Osipov, "Inducing Suffix and LCP Arrays in External Memory," in *Proceedings of the Meeting on Algorithm Engineering and Experiments*, Louisiana, USA, January 2013, pp. 88–102.

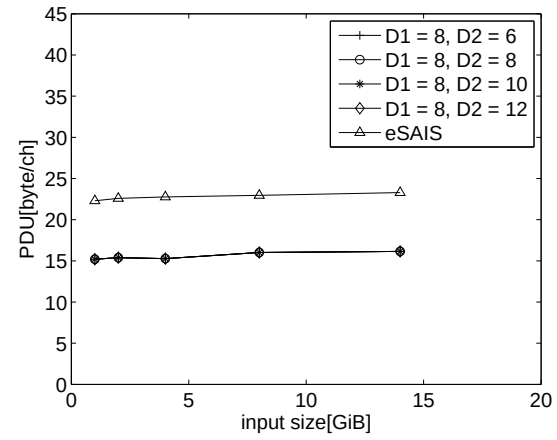
[8] G. Nong, W. H. Chan, S. Zhang, and X. F. Guan, "Suffix Array Construction in External Memory Using D-Critical Substrings," *ACM Transactions on Information Systems*, vol. 32, no. 1, pp. 1:1–1:15, January 2014.

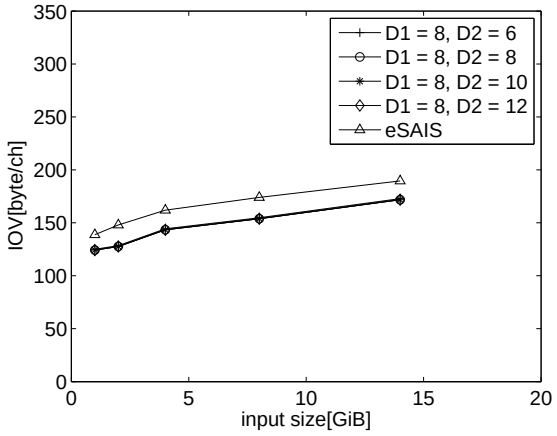
[9] G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu, "Induced Sorting Suffixes in External Memory," *ACM Transactions on Information Systems*, vol. 33, no. 3, pp. 12:1–12:15, March 2015.

[10] G. Nong, S. Zhang, and S. Chan, "Two Efficient Algorithms for Linear Time Suffix Array Construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, 2011.

[11] S. Burkhardt and J. Kärkkäinen, "Fast lightweight suffix array construction and checking," in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Mexico, June 2003, pp. 25–27.

[12] 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.





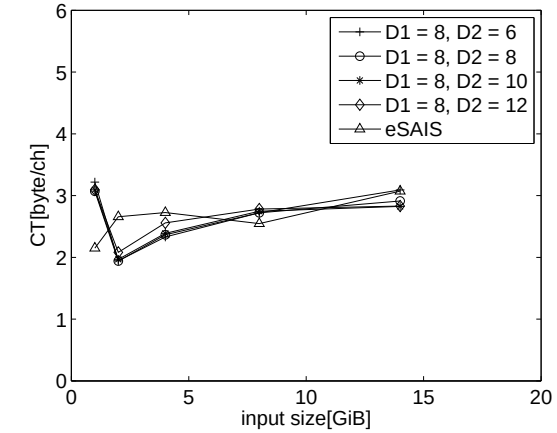


Fig. 3. Experimental results for DSA-IS+ and eSAIS on guten in terms of peak disk usage, I/O volume and construction time, where $D_1 = 8$, D_2 ranges in {6, 8, 10, 12} and the input size varies in {1, 2, 4, 8, 14} GiB.