Induced Sorting Suffixes in External Memory

GE NONG, Sun Yat-sen University, SYSU-CMU Shunde International Joint Research Institute

WAI HONG CHAN, The Hong Kong Institute of Education

SHENG QING HU, Sun Yat-sen University

YI WU, Sun Yat-sen University

We present in this article an external memory algorithm, called disk SA-IS (DSA-IS), to exactly emulate the induced sorting algorithm SA-IS previously proposed for sorting suffixes in RAM. DSA-IS is a new disk-friendly method for sequentially retrieving the preceding character of a sorted suffix to induce the order of the preceding suffix. For a size-n string of a constant or integer alphabet, given the RAM capacity $\Omega((nW)^{1/5})$, where W is the size of each I/O buffer that is large enough to amortize the overhead of each access to disk, both the CPU time and peak disk use of DSA-IS are $O(n)$. Our experimental study shows that on average, DSA-IS achieves the best time and space results of all of the existing external memory algorithms based on the induced sorting principle.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Non-numerical Algorithms and Problems—Sorting and searching; G.2.1 [Discrete Mathematics]: Combinatorics—Combinatorial algorithms; H.3.4 [Information Storage and Retrieval]: Systems and Software—Performance evaluation (efficiency and effectiveness)

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Suffix array, sorting algorithm, external memory

ACM Reference Format:

1. INTRODUCTION

For a size-n input string $x[0,n-1]$ over an ordered alphabet of characters in $[0,\sigma-1]$, the substring starting from $x[i]$ and running to $x[n-1]$ is called a suffix and denoted by $suf(x,i)$. The problem of sorting suffixes is to lexicographically sort all of the suffixes of $x$ into increasing order. The result of sorting is generally stored in an integer array $sa[0,n-1]$, called a suffix array [Manber and Myers 1993], in which each item is a log n-bit integer storing the index for the start position of a suffix in $x$. Hence, sorting all of the suffixes of $x$ is also widely known as constructing the suffix array of $x$.

The suffix array is a fundamental data structure in many applications and its construction, in either internal memory or external memory, is crucial to the applications’
overall efficiencies. The recent book by Ohlebusch [Ohlebusch 2013] has a sophisticated survey of the applications of enhanced suffix arrays (the original suffix array enhanced by other auxiliary data structures such as lcp-table) in bioinformatics. Suffix arrays are used in many information processing applications, and their construction is a problem that has received intensive research attention.

Many suffix array construction algorithms (SACAs) have been proposed on random-access machine models. Puglisi [Puglisi et al. 2007] provide a comprehensive survey up to 2007. Recently, work has been done on external memory SACAs. Recent experimental results have shown that algorithms based on the induced sorting principle, i.e., eSAIS [Bingmann et al. 2013] and EM-SA-DS [Nong et al. 2014], are competitive against the previously best techniques such as DC3 [Dementiev et al. 2008] and bwt-disk [Ferragina et al. 2012]1.

Let $SA(x)$ denote the suffix array of $x$. The key steps to compute $SA(x)$ by induced sorting are: (1) reduce $x$ to the string $x_1[0, n_1 − 1]$ with $n_1 ≤ [n/2]$; (2) compute $SA(x_1)$; and (3) induce $SA(x)$ from $SA(x_1)$. The time complexity of induced sorting is linear as given by the recursive formula $T(n) = T(n/2) + O(n) = O(n)$. The induced sorting method also facilitates time and space efficient designs in practice, e.g., SA-IS [Nong et al. 2011], its optimized implementation [Mori 2008] and a recent improvement of SA-IS called SACA-K, which uses only a $σ \log n$-bit workspace beyond the input string and the output suffix array [Nong 2013].

If both $x$ and $SA(x)$ are held completely in the RAM, the process of inducing $SA(x)$ from $SA(x_1)$ consists of two scans of $SA(x)$ [Nong et al. 2011]. The L-type suffixes are sorted from the sorted L-type (LMS) suffixes and the S-type suffixes are sorted from the sorted L-type suffixes (refer to Section 2.1 for the definitions of L-type, S-type and LMS suffixes). The details are described below, in which $bkt[0, σ − 1]$ is an integer array.

- Inducing L-type suffixes: Scan $sa$ from left to right, $j = sa[i] − 1$ for $i$ increased from 0 to $n − 1$. If $j ≥ 0$ and $x[j]$ is L-type, put $suf(x, j)$ into the current head item of its bucket in $sa$ by setting $sa[bkt[x[j]] + +] = j$ and shift the bucket head one place to the right.

- Inducing S-type suffixes: Scan $sa$ from right to left, $j = sa[i] − 1$ for $i$ decreased from $n − 1$ to 0. If $j ≥ 0$ and $x[j]$ is S-type, put $suf(x, j)$ into the current end item of its bucket in $sa$ by setting $sa[bkt[x[j]] − −] = j$ and shift the bucket end one place to the left.

These two passes are symmetrically analogous. If $x$, $sa$ and $bkt$ are completely stored in the RAM, these two steps can be done very quickly. However, random accesses of $x[j]$, $bkt[x[j]]$ and $sa[bkt[x[j]]]$ in the external memory are problematic, because they lead to slow disk seeks. Thus, a version of SA-IS that runs in the external memory should be developed to avoid these random accesses. Retrieving $x[j]$ without random accesses of $x$ is still a challenge in the existing external memory designs for induced sorting suffixes. The attempts to solve this problem in [Bingmann et al. 2013] and [Nong et al. 2014] resulted in much more complicated designs than their counterparts in the RAM.

Following our recent work [Nong et al. 2014] developing an external memory design for the algorithm SA-DS [Nong et al. 2011], we propose a new solution called disk SA-IS (DSA-IS) in this paper. Pleasingly, unlike the solutions in [Bingmann et al. 2013] and [Nong et al. 2014], DSA-IS emulates SA-IS merely by replacing random accesses of the RAM with sequential accesses of the external memory. We develop DSA-IS on the same settings as the external memory model in [Nong et al. 2014] with the following memory parameters (in $\log n$-bit words): RAM capacity $M = Ω((nW)^{0.5})$, disk capacity

---

1bwt-disk computes the Burrows-Wheeler transform (BWT) and the others construct the suffix array.
$E = O(n)$ and the size of each I/O buffer $W$. $W$ is large enough to amortize the overhead of each disk access; given $W$, the I/O complexity in terms of the number of I/Os is defined as the total I/O volume over $W$.

This work has a number of motivations. The suffix array is an indispensable data structure for many applications such as full-text retrieval and sequence alignment. In all applications involving a suffix array, constructing the suffix array is the first, crucial step. Induced sorting is commonly recognized as a promising method for suffix array construction in both the internal memory and external memory. Two external memory algorithms, eSAIS and EM-SA-DS, have been designed based on induced sorting. However, their designs are complicated and challenging for engineering by third-party users, and their time-and-space performance must be further improved. For such a fundamental problem, a better solution with simpler design and better performance is preferred and desired.

In this article, Section 2 gives the preliminaries for presenting DSA-IS, Section 3 the algorithm’s details, Section 4 the experiments for performance evaluation and Section 5 the summary.

2. OVERVIEW

2.1. Notations

Some basic notations for induced sorting are necessary for presenting this work:

**L-type, S-type and LMS suffixes/characters.** The suffixes in $x$ are classified into two classes: $\text{su}(x, i)$ is S-type if (1) $i = n - 1$, or (2) $\text{su}(x, i) < \text{su}(x, i + 1)$ for $i \in [0, n - 2]$; otherwise, $\text{su}(x, i)$ is L-type. Moreover, $\text{su}(x, i)$ is LMS if $\text{su}(x, i) < \text{su}(x, i - 1)$ is L-type for $i \in [1, n - 1]$. A suffix and its head character are considered to be the same type: $x[i]$ is said to be L-type, S-type or LMS if $\text{su}(x, i)$ is L-type, S-type or LMS, respectively.

**L-type, S-type and LMS substrings.** For all $i < j$, if $x[j]$ is LMS and there is no LMS character in $x[i + 1, j - 1]$, then $x[i, j]$ is L-type, S-type or LMS as long as $x[i]$ is L-type, S-type or LMS, respectively. Moreover, $x[n - 1]$ itself is an S-type and an LMS substring.

**Bucket.** In $\text{SA}(x)$, all of the suffixes with the same head character, $ch$, are consecutively stored in a range called the bucket for $ch$, denoted by $\text{bucket}(\text{SA}(x), ch)$. The leftmost and the rightmost items of a bucket are called the start and the end items of the bucket, respectively. If there are both L-type and S-type characters in a bucket, all of the L-type characters are on the left side of the bucket and all of the S-type on the right.

**Preceding character.** The preceding character of $x[i]$ for $i \in [0, n - 1]$, denoted by $\text{prec}(x, i)$, is $x[i - 1]$ for $i > 0$, or else $x[n - 1]$. The preceding character of a suffix/substring starting at $x[i]$ is also $\text{prec}(x, i)$.

**Reduced string.** The reduced string $x_1$ is formed by replacing all of the LMS substrings in $x$ with their integer names, which represent the order of the LMS substrings.

We assume that $x[n - 1] = 0$ is the unique smallest character of $x$ and that each character is equal to the number of characters in $x$ smaller than it. Under these assumptions, $x[n - 1]$ is usually called the sentinel. It guarantees that any suffix is unique and not a prefix of another. $x[i]$ itself also gives the start position of $\text{bucket}(\text{SA}(x), x[i])$. These assumptions are rather easy to satisfy in practice\(^2\).

\(^2\)For an input string $x$ of $O(1)$ or $n^{O(1)}$, all of the characters of $x$ are sorted by an external memory integer sorting algorithm and each character is renamed as the number of characters smaller than it. It is
2.2. DSA

For $SA(x)$, we introduce its external memory alternative $DSA(x)$ (i.e., disk-SA for $x$). Each item of $DSA(x)$ is a tuple $⟨pos, ch, t⟩$ called DSAITEM, where each symbol is defined as:

- $pos$: position index for $suf(x, pos)$.
- $ch$: $x[pos]$, i.e., the head character of $suf(x, pos)$.
- $t$: 0 or 1 for $prec(x, pos)$ being L-type or S-type, respectively.

For an instance $e$ of DSAITEM, we say that $e$ stores a suffix/substring in $x$ if $e.pos$ is the start position of the suffix/substring in $x$.

Let $DSA(x)$ and $DSAB(x)$ be two size-$n$ arrays of DSAITEM. $DSA(x)$ is defined as $DSA(x)[i].pos = SA(x)[i]$ and $DSA(x)[i].⟨ch, t⟩$, set as specified in DSAITEM. Using $DSA(x)$, we define the disk BWT of $x$, denoted by $DSAB(x)$, as $DSAB(x)[i].pos$ set as $DSA(x)[i].pos - 1$ or $n - 1$, for $DSA(x)[i].pos$ greater than or equal to 0, respectively, and $DSAB(x)[i].⟨ch, t⟩$ set as specified in DSAITEM.

The key idea behind the design of DSA-IS is to split $x$ into blocks and divide $DSA(x)$ and $DSAB(x)$ for each block of $x$. Compute, in the RAM, $DSA(x)$ and $DSAB(x)$ of each block in a block-by-block manner. Compute $DSA(x)$ by sequentially accessing the data of $DSA(x)$ and $DSAB(x)$ in each block via an I/O buffer of $W$ words per block. SA-IS is used to compute in the RAM the external representation $DSA(x,b_i|x)$ of the suffix array of each block $x,b_i$ of $x$. The whole suffix array of $x$ can then be efficiently induced by merging all of the $DSA(x,b_i|x)$ together in a scanning complexity. Prior to the merging, each $DSA(x,b_i|x)$ is augmented with the BWT of $x,b_i$, i.e., $DSAB(x,b_i|x)$, to enable fast sequential access to the preceding character of each sorted suffix in $x,b_i$ during the inducing process, where a suffix $suf(x, j)$ is said to be in $x,b_i$ if $x[j] \in x,b_i$.

2.3. Dividing $x$ and $dsa$ into Blocks

Below are the general constraints for dividing $x$ into $n/m$, where $m = O(M)$, consecutive blocks $\{x,b_i|i \in [0, k-1]\}$:

(1) Each block starts with $x[0]$ or an LMS character and ends with another LMS character.

(2) Any pair of neighboring blocks overlap on a common LMS character.

(3) A block $x[g,h]$, $0 \leq g < h \leq n-1$, can have more than $m$ characters only if there is no LMS character in $x[g+1,h-1]$.

There are many ways to divide $x$ under the above constraints. Here we initialize $i = k - 1$ and $x,b_i$ as empty. For all of the LMS substrings from right to left in $x$, a substring will be added to $x,b_i$ if the addition will not cause $|x,b_i| > m$, or else $i$ is decreased by 1 and a new block, $x,b_i$, consisting of this substring at its rightmost is created. This block division can be done by scanning $x$ leftwards. Similarly, we can also divide $x$ by scanning $x$ rightwards, but this will require a somewhat complicated method to check the type of each character to on-the-fly detect each substring, and hence is not used here.

The constraints and the division strategy lead to $\Sigma|x,b_i| = n + k - 1$ and $|x,b_i| + |x,b_{i+1}| \geq m + 2$ for $i \in [0, k-2]$. Thus, $(k - 1)(m + 2) < n + k - 1$ and hence $k < \frac{2m}{m + 1}$. The number of blocks does not exceed $\lceil \frac{2m}{m + 1} \rceil$.

For $i \in [0, k-1], x[i \cdot m]$ is called a boundary character. Each character $x[j], i \cdot m \leq j \leq (i + 1) \cdot m$ must belong to the block containing $x[i \cdot m]$, the block containing $x[(i + 1) \cdot m]$
or the block in between these two. The three blocks may be the same or all different. Hence, for each boundary character, we record the start and end positions of the block containing it. In this way, given the position of a character, we can locate the block containing this character in \(O(1)\) time and \(O(k)\) space.

In addition to dividing the input \(x\) into blocks, the output \(dsa[0, n-1]\) of \(n\) \(DSAITEM\) tuples is also divided into blocks. We split \(dsa\) into \(dsa_k\) blocks \(\{dsa_{bh} | i \in [0, dsa_k - 1]\}\) evenly (except that the last block may be smaller), where \(dsa_{bh} = dsa[i \cdot dsa_m, (i + 1) \cdot dsa_m - 1]\), \(dsa_k = \lceil n/dsa_m \rceil\) and \(dsa_m = O(M)\). It should be noted that \(m\) and \(dsa_m\) and are two independent parameters for the division of \(x\) and \(dsa\), respectively, under the given RAM limit. They are generally different for optimal performance; nevertheless, they can also be set as identical for engineering convenience in practice.

### 2.4. Algorithm Framework

The framework of DSA-IS remains similar to that of SA-IS and is sketched below:

```plaintext
DSA-IS(x) {
    /* Reducing the Problem */
    compute the reduced string \(x_1\) by Algorithm II in Section 3.2;
    /* Recursion */
    if (the suffix array of \(x_1\) can be computed in RAM)
        compute DSA(\(x_1\)) in RAM by the algorithm SA-IS.
    else
        DSA-IS(\(x_1\));
    /* Inducing the Solution */
    compute DSA(\(x\)) from DSA(\(x_1\)) by Algorithm I in Section 3.1;
}
```

Similar to SA-IS in RAM, with slight modifications, the algorithm for induced sorting of suffixes in the external memory can also be reused for sorting LMS substrings. Hence, the sorting algorithm for inducing the solution is described first. It is used to develop the algorithm for sorting and naming LMS substrings to reduce the problem.

### 3. ALGORITHM DETAILS

Radix sort is frequently used in DSA-IS to sort fixed-size items of integer keys. Given \(M = \Omega((nW)^{0.5})\) in our external memory model, the sorting of each \(\log n\)-bit integer can be done in two passes using a multi-pass radix sort. The first pass sorts the lowest \(0.5 \log n\) bits and the second pass sorts the highest \(0.5 \log n\) bits.

#### 3.1. Inducing Solution

We define the following sets for \(x_{bh} = x[g, h]\) with \(g < h\) and \(i \in [0, k - 1]\), where the symbol "|" (i.e., on condition of \(x\)) in each set indicates that the ordering of the suffixes in the set may be affected by parts of \(x\) outside \(x_{bh}:

- \(SA_{lms}(x_{bh}|x) = \{SA(x)[j] | SA(x)[j] \in [g + 1, h]\} \) and \(x[SA(x)[j]] \) is LMS, \(j \in [0, n - 1]\).
- \(DSA(x_{bh}|x) = \{DSA(x)[j] | DSA(x)[j], pos \in [g, h - 1], j \in [0, n - 1]\}\).
- \(DSAB(x_{bh}|x) = \{DSAB(x)[j] | DSA(x)[j], pos \in [g, h - 1], j \in [0, n - 1]\}\).

Using \(x_{bh}\) and \(SA_{lms}(x_{bh}|x)\) is a key issue in inducing the solution. We make two important observations with respect to whether \(x_{bh}\) is a single-substring block or not.

If \(x_{bh}\) is not a single-substring block, the size of \(x_{bh}\) is at most \(m\) and hence both \(x_{bh}\) and \(SA_{lms}(x_{bh}|x)\) can be stored in the RAM. The characters in \(x_{bh}\) are sorted and
renamed as their ranks starting from 0 to form a new block \(x_{j'}\). The LMS suffix array of \(x_{j'}\), i.e., \(SA_{lms}(x_{j'} \mid x)\), can be derived from \(SA_{lms}(x_{j} \mid x)\) trivially. Then \(DSA(x_{j} \mid x)\) can be computed from \(x_{j'}, x_{j'}',\) and \(SA_{lms}(x_{j'} \mid x)\) using a method similar to that for inducing \(SA(x)\) from \(SA(x_1)\) as explained in the introduction. Hence, we have the first observation.

**Observation 3.1.** Given \(x_{j}\) and \(SA_{lms}(x_{j} \mid x)\), where \(x_{j}\) is not a single-substring block, \(DSA(x_{j} \mid x)\) can be computed in linear CPU time and the RAM space \(O(x.n_i)\), by adapting the algorithm for induced sorting in SA-IS with minor modifications.

When \(x_{j}\) is a single-substring block, the size of \(x_{j}\) may exceed \(m\) and hence it cannot be put in the RAM. We thus make the second observation as follows.

**Observation 3.2.** \(DSA(x_{j} \mid x)\) for a single-substring block \(x_{j}\) can be computed in the external memory.

In a single-substring block \(x_{j}\), all of the L-type and S-type suffixes in the block are already sorted from right to left into their increasing and decreasing orders, respectively. \(DSA(x_{j} \mid x)\) can be computed by scanning and merging these two kinds of sorted suffixes in CPU time \(O(x.n_i)\) and I/O complexity \(O(x.n_i/W)\), where \(x.n_i = ||x_{j}||\).

As an example for demonstrating how to sort the suffixes in a single-substring block, let us suppose that \(x_{j} = \text{"abcce...xyyzzz...cba"}\) is a single-substring block with all of the S-type characters marked in bold. Notice that the last suffix in \(x_{j}\) is not contained in \(DSA(x_{j} \mid x)\). To compute \(DSA(x_{j} \mid x)\), we use two I/O buffers to sequentially retrieve the L-type and S-type suffixes (excluding the last S-type suffix) in ascending order, respectively. For any two suffixes being compared for merging, their order can be determined immediately if their head characters are different; otherwise, the S-type suffix must be greater (see Lemma 2 in [Ko and Aluru 2005]). For instance, in this example of the two suffixes starting with “y” in \(x_{j}\), the one with a bold “y” is greater.

With these observations, the algorithm for induced sorting is given below:

---

**Algorithm I: Inducing a Solution in the External Memory**

- **Input:** \(x\), \(DSA(x_1)\).
- **Output:** \(DSA(x)\).
- **Procedure:**
  1. Compute \(SA_{lms}(x_{j} \mid x)\) from \(DSA(x_1)\) for all \(i \in [0, k - 1]\).
  2. Make \(DSA(x_{j} \mid x), DSA_{L}(x_{j} \mid x)\) and \(DSA_{S}(x_{j} \mid x)\) from \(SA_{lms}(x_{j} \mid x)\) for all \(i\).
  3. Merge \(DSA(x_{j} \mid x)\) for all \(i\) to produce \(DSA(x)\).

---

More details are given below to each step in Algorithm I.

**Step I.1**

This step consists of three sub-steps:

(a) Radix sort \(j\) by \(DSA(x_1)[j].pos\) for all \(j \in [0, n_i - 1]\) to produce \(ISA(x_1)\), i.e., the inverse suffix array of \(x_1\) defined as \(ISA(x_1)[j] = j\) for all \(j\).

(b) Radix sort \((PA_{lms}(x)[j], ch, t)\) by \(ISA(x_1)[j]\) for all \(j\). The sorting result is \(ISA_{lms}(x)\), which contains all of the sorted LMS suffixes in \(x\), where \(PA_{lms}(x)\) is the position array for all of the LMS characters in \(x\). \(PA_{lms}(x)[j]\) gives the starting position of the \(j\)th LMS character in \(x\).

(c) Scan and decompose \(ISA_{lms}(x)\) into \(SA_{lms}(x_{j} \mid x)\) for all \(i \in [0, k - 1]\) as follows. For each \(ISA_{lms}(x)[j]\) being scanned, denoted by \(e\), we can determine in \(O(1)\) time and \(O(k)\) space which \(x_{j}\) contains \(x[e.pos]\). Once \(SA_{lms}(x_{j} \mid x)\) is located, \(e.pos\) is
appended to the I/O buffer for $SA_{ims}(x,b_i|x)$. The I/O buffer will be flushed to disk when it is full.

Step I.2
There are two cases with respect to whether $x,b_i$ is a single-substring block or not for computing $DSA(x,b_i|x)$ from $SA_{ims}(x,b_i|x)$.

— Yes: $DSA(x,b_i|x)$ can be computed in the disk (see Observation 3.2).
— No: $DSA(x,b_i|x)$ can be computed in the RAM (see Observation 3.1), because $x.n_i \leq m$.

Then, $DSAB_x(x,b_i|x)$ and $DSAB_s(x,b_i|x)$ are computed by scanning $DSA(x,b_i|x)$.

Step I.3
This step consists of three sub-steps, in which random access of $x[p]$ in Sub-steps (b) and (c) is avoided because $e_1.ch = x[p]$;

(a) Initialize each item of $dsa$ as empty by setting all of its members to 0. Scan $DA_{ims}(x)$ leftwards. For each scanned item $e$, let $p = e.pos$ and put $suf(x,p)$ in the current rightmost empty item in bucket($dsa,x[p]$), say $dsa[j]$, by copying $e$ to $dsa[j]$.

(b) Scan $dsa$ rightwards. For each scanned item $e$ with $e.pos > 0$ and $e.t = 0$, let $p = e.pos - 1$. Determine the block in $x$ containing $x[e.pos]$, say $x,b_r$. Put $suf(x,p)$ in the current leftmost empty item in bucket($dsa,x[p]$), say $dsa[j]$, by copying the current leftmost unvisited item in $DSAB_x(x,b_r|x)$, say $e_1$, to $dsa[j]$ and mark $e_1$ as visited.

(c) Scan $dsa$ leftwards. For each scanned item $e$ with $e.pos > 0$ and $e.t = 1$, let $p = e.pos - 1$. Determine the block in $x$ containing $x[e.pos]$, say $x,b_l$. Put $suf(x,p)$ in the current rightmost empty item in bucket($dsa,x[p]$), say $dsa[j]$, by copying the current rightmost unvisited item in $DSAB_s(x,b_l|x)$, say $e_1$, to $dsa[j]$ and mark $e_1$ as visited.

Avoiding Random Access of $dsa[j]$ in Step I.3
In Algorithm I, all of the steps except Step I.3, adapt easily to disk. The obstacle for making an external memory design for Step I.3 is the need to “put $suf(x,p)$ in the current leftmost/rightmost empty item in bucket($dsa,x[p]$),” $dsa[j]$ may be currently on disk and will be accessed slowly. To overcome this difficulty, the blockwise induced sorting method reported in [Beller et al. 2013] and [Nong et al. 2014] can be applied to avoid random access of $dsa[j]$.

Blockwise induced sorting evenly splits $dsa$ into a number of blocks (the last block may be smaller). It repeats the following steps for all of the blocks of $dsa$ sequentially (from left to right for inducing L-type suffixes and from right to left for inducing S-type suffixes), where the $dsa$ block currently in the RAM is the active $dsa$ block.

1. After a block in $dsa$ has been switched from the disk to the RAM and has become the active block, the suffixes already in the block are stably radix sorted to the correct positions in their buckets by their head characters. In each bucket, the L-type and S-type suffixes are clustered at the left end and right end, respectively.

2. For each suffix whose order is being induced from a sorted suffix in the active block, determine the $dsa$ block, say $dsa,b_i$, that the suffix should be sorted into and put the suffix into $dsa,b_i$ as follows: if $dsa,b_i$ is the active $dsa$ block, put the suffix in its correct position in $dsa,b_i$, or else append the suffix to the I/O buffer for $dsa,b_i$.

Fig. 1 shows the main data structures used by the blockwise induced sorting algorithm with $x$ and $dsa$ divided into three and four blocks, respectively. In this example, block 1 of $x$ contains only a single substring and is much longer than the other two blocks, whereas all of the blocks of $dsa$ are the same length.
A:8 Induced Sorting Suffixes in External Memory

Fig. 1: The main data structures used in the blockwise induced sorting method, where “DSAB blocks” include \{DSAB_ℓ(x,b_i|x)\} and \{DSAB_s(x,b_i|x)\} for all i for inducing L-type and S-type suffixes, respectively, in Steps I.3.b and I.3.c of Algorithm I. The I/O buffers and active dsa block are stored in the RAM, and DSAB and dsa are in the disk.

3.2. Reducing the Problem

The main task for reducing the size of the problem is to sort and name all of the LMS substrings. By replacing each substring in \(x\) with its new name, a new shorter string can be produced. A naive method for sorting substrings in the external memory is to use a \(k\)-way mergesort: sort all of the substrings in each block \(x_b_i\) and merge all of the sorting results. This mergesort requires \(O(n \log k)\) CPU time. However, a mergesort works only when all of the \(k\) substrings in a comparison for merging are available in the RAM. We should avoid using a mergesort for reducing the size of the problem in our solution, because the \(k\) substrings may require storage exceeding \(M\) and each substring may need to be read from the disk many times for comparisons. We reuse the method for induced sorting of suffixes in Step I.3 of Algorithm I, similar to what we did for sorting substrings in SA-IS.

Before presenting the algorithm for reducing the size of the problem, we first define some symbols on \(x\):

- \(\text{DSTR}(x)\): a size-\(n\) DSAITEM array, storing all of the L-type and S-type substrings sorted in lexicographical order.
- \(\text{DSTRB}(x)\): \(\text{DSTRB}(x)[i].pos = j\) and \(\text{DSTRB}(x)[i].\{ch,t\}\) are set as specified in the definition of DSAITEM, where \(j = \text{DSTR}(x)[i].pos - 1\) for \(\text{DSTR}(x)[i].pos > 0\) or else \(j = n - 1\), for \(i \in [0, n - 1]\).
- \(\text{DSTR}_{\text{lms}}(x) = \{\text{DSTR}(x)[i].\text{pos}[x|\text{DSTR}(x)[i].\text{pos}] \text{ is LMS}, i \in [0, n - 1]\}\).

Next, we let \(x_b_i = x[g,h]\) with \(g < h\) for all \(i \in [0,k-1]\) and define some more symbols on \(x,b_i\):

- \(\text{DSTR}(x,b_i|x) = \{\text{DSTR}(x)[j]|\text{DSTR}(x)[j].\text{pos} \in [g,h - 1], j \in [0,n - 1]\}\).
- \(\text{DSTRB}(x,b_i|x) = \{\text{DSTRB}(x)[j]|\text{DSTR}(x)[j].\text{pos} \in [g,h - 1], j \in [0,n - 1]\}\).
— DSTRBℓ(x.bℓ|x) and DSTRBℓ(x.br|x) contain only the items of the L-type and S-type substrings in DSTRB(x.bℓ|x), respectively.
— DSTRDATAℓms(x.bℓ|x) stores the characters for the sorted LMS substrings in x.bℓ, i.e., each item of DSTRDATAℓms(x.bℓ|x) is a substring (not an index to the substring).

Algorithm II: Reducing the Problem in the External Memory

— Input: x.
— Output: x₁.
— Procedure:
  (1) Compute DSTR(x.b₁|x) and DSTRDATAℓms(x.b₁|x) for i ∈ [0, k − 1].
  (2) Merge DSTR(x.b₁|x) for all i to produce DSTRℓms(x).
  (3) Scan DSTRℓms(x) and DSTRDATAℓms(x.b₁|x) for all i to compute the name for each LMS substring in x, using DSTRℓms(x) as the pivot for the order of sequentially retrieved substrings from each DSTRDATAℓms(x.b₁|x).
  (4) Replace each LMS substring in x by its name to produce the reduced string x₁.

More details of each step in Algorithm II are given below.

Step II.1
Compute DSTR(x.b₁|x) using the algorithm for sorting LMS substrings in SA-IS and scan DSTR(x.b₁|x) to produce DSTRBℓ(x.b₁|x), DSTRBℓ(x.br|x) and DSTRDATAℓms(x.b₁|x).

Step II.2
This step consists of three sub-steps:

(a) Scan x leftwards to put all of the LMS suffixes into their buckets in dsa, from right to left in each bucket.
(b) Reuse Steps 1.3.b and 1.3.c of Algorithm I to compute DSTR(x) in the disk by replacing DSABℓ(x.br|x) and DSABℓ(x.b₁|x) with DSTRBℓ(x.b₁|x) and DSTRBℓ(x.br|x), respectively.
(c) Scan DSTR(x) to produce DSTRℓms(x).

In Sub-step (c) above, we need to detect if a character x[DSTR(x)[i].pos] is LMS or not. This can be done trivially as follows. In our program, each bucket in DSTR(x) is split into the L-type and the S-type sub-buckets for all of the L-type and S-type items respectively, in this bucket. All of the L-type sub-buckets in DSTR(x) are consecutively stored in a file and similarly all of the S-type sub-buckets in another file. When DSTR(x)[i] is in an S-type sub-bucket, we retrieve the type of its preceding character from DSTR(x)[i].ℓ to make a decision.

Step II.3
Scan DSTRℓms(x) rightwards. For each item e being scanned, let x.bh be the block containing x[e]. The LMS substring starting at x[e] must be the current leftmost unvisited substring in DSTRDATAℓms(x.bh|x). The sorted LMS substrings are sequentially retrieved and marked as visited from DSTRDATAℓms(x.bh|x), for all i. Hence, any pair of neighboring substrings in DSTRℓms(x) can be sequentially retrieved and compared once to determine if they are equal or not. The name of each substring in DSTRℓms(x), which is defined as the number of all of the substrings less than it, can also be computed.
Step II.4

DNAME_{lms}(x) is an integer array storing the names of all of the sorted LMS substrings in DSTR_{lms}(x), i.e., DNAME_{lms}(x)[i] is the name for the LMS substring starting at x[DSTR_{lms}(x)[i].pos]. The reduced string x_1 is produced by radix sorting DNAME_{lms}(x)[i] by DSTR_{lms}(x)[i] for all i.

3.3. Analysis

The time and space complexities of DSA-IS are dominated by Algorithm I for induced sorting; hence it suffices to conduct the analysis on Algorithm I.

The maximum RAM requirement is for storing the data structures shown in Fig. 1, where all of the I/O buffers for the blocks of DSAB and dsa must be simultaneously maintained in the RAM. Thus the maximum number of blocks is \((k+dsa,k) \cdot W = O(M)\), i.e., \(k + dsa,k = O(M/W)\). As \(dsa,m = O(M)\), the maximum input size is \(n = dsa,k \cdot dsa,m \leq (k + dsa,k) \cdot dsa,m = O(M/W) \cdot O(M) = O(M^2/W)\), and so \(M = \Omega((nW)^{0.5})\). This meets the assumptions for our external memory model given in Section 1. With these I/O buffers, accessing the data of each DSAB or dsa block is done in an I/O complexity of the block's size over \(W\), yielding the algorithm's total I/O complexity as \(O(n/W)\).

DSA-IS thus exactly emulates SA-IS and, therefore, inherits the advantages of the latter: both the CPU time and peak disk use are \(O(n)\) and the I/O complexity is \(O(n/W)\).

4. EXPERIMENTS

The time and space performance of DSA-IS are evaluated by comparing with eSAIS [Bingmann et al. 2013] and EM-SA-DS [Nong et al. 2014] on the datasets listed in Table I. All of the datasets except the last one were used in the experimental study in [Nong et al. 2014] for evaluating the performance of EM-SA-DS. The last dataset is a recent version of the Gutenberg collection. It is chosen not only because it is large enough for our experiments, but also because its alphabet size of 256 is typical for texts in full ASCII.

Our experimental platform has an identical configuration to that used in the experimental study in [Nong et al. 2014]: 1 CPU (Intel(R) Core(TM) i3 3.20 GHz); 4 GiB RAM (1333 MHz DDR3); 1 Disk (2 TiB, 7200 rpm, SATA2); Linux (Ubuntu 11.04). The program for eSAIS is downloaded from the web page\(^3\) described in [Bingmann et al. 2013] and the programs for EM-SA-DS and DSA-IS are our own implementations. A package containing our programs and SACA-K can be retrieved from our project site\(^4\). For the convenience of presentation, these programs are denoted by the lowercase symbols esais, emsads, dsais and sacak, respectively.

Two experiments are conducted to measure the time and space consumptions of each algorithm. The program for each algorithm is set 3 GiB RAM and 40-bit integers. The performance metrics measured are the mean speed in microseconds per character (\(\mu s/ch\)), peak RAM use in Giga bytes, peak disk use and mean I/O volume in bytes per character, and the recursion depth. To obtain the statistics for each program, the running time and peak RAM use are collected using the shell commands “time” and “memusage,” respectively, and the other metrics are collected by the program itself. For accurate time results, the time of each program on a corpus is evaluated as the mean of 2 runs.

For the recursion depth, esais counts all of the recursions as in SA-IS, i.e., the deepest recursion level considers all of the characters of the reduced string to be different.

---

\(^3\)http://panthema.net/2012/1119-eSAIS-Inducing-Suffix-and-LCP-Arrays-in-External-Memory/.

\(^4\)http://code.google.com/p/ge-nong/
Table I: Corpora, \( n \) in GiB, 1 byte per character.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>( n )</th>
<th>( \sigma )</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>2.86</td>
<td>6</td>
<td>Human genome data, used in [Dementiev et al. 2008], at <a href="http://algo2.iti.kit.edu/dementiev/esuffix/instances">http://algo2.iti.kit.edu/dementiev/esuffix/instances</a>.</td>
</tr>
<tr>
<td>guten</td>
<td>3.05</td>
<td>256</td>
<td>Gutenberg collection, used in [Dementiev et al. 2008], at <a href="http://algo2.iti.kit.edu/dementiev/esuffix/instances">http://algo2.iti.kit.edu/dementiev/esuffix/instances</a>.</td>
</tr>
<tr>
<td>random2</td>
<td>4.00</td>
<td>256</td>
<td>A concatenation of two identical copies of a string with each character randomly selected from ([0, 255]), with a maximum LCP of 2.0 Gi. The exact size of this file is (2^{32} - 2) bytes.</td>
</tr>
<tr>
<td>genome2</td>
<td>5.72</td>
<td>6</td>
<td>A concatenation of two copies of a corpus “genome,” with a maximum LCP of 2.86 Gi.</td>
</tr>
</tbody>
</table>

However, both emsads and dsais count only the recursions down to the recursion level where the suffix array of the reduced string can be computed in the RAM by SA-IS. Hence, the metrics for esais should not be compared with the metrics of the other two. The purpose of collecting these metrics is to see how many recursions are needed for each individual program running on a corpus and to gain more information about the program’s behavior.

4.1. Experiment I: Different Input Data

This experiment investigates the time and space performance of each algorithm on different input data of varying size, alphabet and LCP. The first six corpora in Table I are used.

As a reference benchmark for the sequential I/O throughput of the machine in use, we use the shell command “time cp” to evaluate the time to copy the file “guten1209.” This file duplication job consists of reading and writing the file once. After three runs, a mean running time of 673 seconds is recorded, yielding a saturated sequential I/O throughput of 0.014 \(\mu s\) per byte, i.e., 68.3 MiB/s. To calculate the speed gap between DSA-IS and its internal memory counterpart SA-IS, we also run sacaz, which is an optimized SA-IS design requiring a \(5n\) space for \(n \leq 2^{32}\), on all 600 MiB prefixes of each corpus to obtain a total running time of 2672 seconds, i.e., 0.607 \(\mu s\) per byte.

Table II shows the experimental results for the running time in \(\mu s/ch\) of each program on a corpus, the mean running time and the I/O throughput of each program on all of the corpora, where the mean running time in \(\mu s/ch\) for each program is the total time averaged over all of the corpora and the I/O throughput in \(\mu s\) per byte is the total time averaged over the total I/O volume. In each row, the best result is marked in bold. The row “norm.” gives the results in the row “mean” normalized by the best result. From this table, we conclude that:

— the mean running times of dsais and esais are more or less the same, although dsais is observed to be marginally faster in most cases. As the speed of an I/O intensive...
### Table II: Running times in $\mu$s/ch.
The mean running time for each program is the total time averaged over all of the corpora. The throughput in $\mu$s per byte is the total time averaged over the total I/O volume.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>esais</th>
<th>emsads</th>
<th>dsais</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniprot</td>
<td>2.998</td>
<td>3.124</td>
<td>2.502</td>
</tr>
<tr>
<td>genome</td>
<td>3.029</td>
<td>3.068</td>
<td>2.550</td>
</tr>
<tr>
<td>guten</td>
<td>3.662</td>
<td>3.900</td>
<td>3.250</td>
</tr>
<tr>
<td>random2</td>
<td>4.037</td>
<td>3.744</td>
<td>3.856</td>
</tr>
<tr>
<td>genome2</td>
<td>3.425</td>
<td>3.308</td>
<td>2.966</td>
</tr>
<tr>
<td>enwiki</td>
<td>3.827</td>
<td>4.610</td>
<td>4.185</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>3.586</td>
<td>3.797</td>
<td>3.418</td>
</tr>
<tr>
<td><strong>norm.</strong></td>
<td>1.049</td>
<td>1.111</td>
<td>1.000</td>
</tr>
</tbody>
</table>

| throughput | 0.019 | 0.019 | 0.022 |

### Table III: Peak RAM in GiB, peak hard-disk (HD) use and mean I/O volume in bytes per input character, and recursion depth (RD). The mean of each metric is the total averaged over all of the corpora.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>esais</th>
<th>emsads</th>
<th>dsais</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniprot</td>
<td>3.3</td>
<td>22.8</td>
<td>157.0</td>
</tr>
<tr>
<td>genome</td>
<td>3.7</td>
<td>22.5</td>
<td>153.2</td>
</tr>
<tr>
<td>guten</td>
<td>3.6</td>
<td>24.7</td>
<td>185.3</td>
</tr>
<tr>
<td>random2</td>
<td>3.2</td>
<td>25.4</td>
<td>201.4</td>
</tr>
<tr>
<td>genome2</td>
<td>3.7</td>
<td>22.6</td>
<td>189.1</td>
</tr>
<tr>
<td>enwiki</td>
<td>3.6</td>
<td>24.4</td>
<td>215.6</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>-</td>
<td>23.3</td>
<td>187.3</td>
</tr>
<tr>
<td><strong>norm.</strong></td>
<td>-</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

The I/O throughput of each program is almost equal to 0.02 $\mu$s per byte, which is $0.014/0.02 = 70\%$ of the system's saturated sequential I/O throughput.

The mean running time of dsais is 3.418 $\mu$s per byte, which is 3.418/$0.007 = 5.6$ times that of saca-k, indicating a speed gap of around six times between the internal and external designs for the induced sorting method.

Table III shows that the best results are achieved by dsais. Specifically,

- dsais has the smallest disk capacity requirement.
- the smallest I/O volume is achieved by dsais.

agree with the ratios of the mean running times for esais and emsads against dsais, i.e., 1.049 and 1.111, respectively, and with the running time being proportional to the I/O volume.

![Graph showing running times in µs/ch for prefixes of “guten1209” in lengths from 1 to 16 GiB.](image)

**Fig. 2:** Running times in µs/ch for prefixes of “guten1209” in lengths from 1 to 16 GiB.

Table IV: Peak RAM in GiB, peak hard-disk (HD) use and mean I/O volume in bytes per input character, and recursion depth (RD), for prefixes of “guten1209” in lengths from 1 to 16 GiB. The mean of each metric is the total of this metric averaged over all of the prefixes.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>esais</th>
<th>emsads</th>
<th>dsais</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>HD</td>
<td>I/O</td>
</tr>
<tr>
<td>1</td>
<td>3.6</td>
<td>22.3</td>
<td>139.1</td>
</tr>
<tr>
<td>2</td>
<td>3.7</td>
<td>22.4</td>
<td>143.5</td>
</tr>
<tr>
<td>4</td>
<td>3.7</td>
<td>22.5</td>
<td>154.1</td>
</tr>
<tr>
<td>8</td>
<td>3.7</td>
<td>23.0</td>
<td>172.5</td>
</tr>
<tr>
<td>16</td>
<td>3.7</td>
<td>23.4</td>
<td>193.1</td>
</tr>
<tr>
<td>mean</td>
<td>-</td>
<td>23.1</td>
<td>178.2</td>
</tr>
<tr>
<td>norm.</td>
<td>-</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**4.2. Experiment II: Increasing the Input Size**

In this experiment, the scalability of each program is investigated by evaluating its time and space performance on the prefixes of “guten1209” in lengths varying from 1 to 16 GiB. The experiment results are shown in Fig. 2 and Table IV. In Fig. 2, the running time of each program is plotted against the prefix length. The curves look analogous: all of them increase at similar rates when the prefix length grows. The curves for esais and dsais are so close that they twist together with only negligible distances. However, there is a noticeable gap between the curve for emsads and the other two curves. This gap is due to the larger I/O volume of emsads, as seen in Table IV.

Table IV shows that the best results for disk use and I/O volume are also achieved by dsais. For each program, the peak disk use remains stable as the prefix length increases. However, the I/O volume grows substantially as the prefix length increases.
For \textsc{emsads} and \textsc{dsais} (\textsc{esais} is not analyzed here because we do not know much about its implementation details), the growth of the I/O volume is mainly due to the use of a merge sort in the programs. In some steps of the algorithms \textsc{em-sa-ds} and \textsc{dsa-is}, there are several tasks requiring sorting of fixed-size tuples of integer keys. Those tasks are done using a merge sort instead of a radix sort in our programs. A merge sort is used in the current revisions of the two programs as its external memory design is much easier for us to code with limited time and manpower. In these two programs, replacing merge sorting with radix sorting to achieve a linear I/O volume is a routine engineering job and can be done with enough programming capacity. Due to the use of a merge sort, given a fixed RAM limit, \( M \) is fixed and hence \( m = O(M) \) is fixed. At each recursion level, \( k = n/m \) becomes larger as \( n \) increases and hence the I/O volume \( O(n \log k) \) also increases.

### 4.3. Implementation Issues

Given \( M \) and \( n \), there are many possible choices for \( m \) and \( W \). In our program, the peak RAM use is estimated as \( 91m \) bytes. Given \( M = 3 \text{ GiB} \) in our experiments, when \( n < 10 \text{ Gi} \), we simply set \( m = M/96 = 2^{25} \), hence \( k < 10 \cdot 2^{30}/m = 5 \cdot 2^9 \) and it is safe to set \( W = (96 - 91)m/k = m/2^6 = 2^{19} \). For bigger \( n \), we choose \( m = M/91 \), and \( W \) is dynamically set as \( m \) divided by the number of I/O buffers. For the program running on the 16 GiB prefix of “guten1209,” these parameters are recorded as: \( W = 71.2 \text{ KiB} \), \( m = \text{dsais} \cdot m \) = 33.8 MiB and \( k = \text{dsais} \cdot k = 486 \). For a given \( M \), if we keep choosing \( m \) and \( W \) in this way, then \( W \) will decrease when \( n \) increases and the I/O complexity \( O(n/W) \) will proportionally increase to slow down the program. To solve this problem, we can choose a smaller \( m \) to save RAM space for the I/O buffers when \( W \) is less than a given threshold. In the current revision of our program, we fix \( m = M/91 \) and use 40-bit integers, resulting in a peak disk use of around \( 20n \) bytes. If we do not take the issue of \( W \) into account, the largest file that \textsc{dsais} can handle on the experiment platform is 2048 GiB/20 > 100 GiB. However, as mentioned above, fixing \( m = M/91 \) in this case is obviously not good enough and a better method must be used for improving the performance.

Our programs \textsc{dsais} and \textsc{emsads} are natural implementations of the algorithms for conducting our experiments. They can be further refined for better performance. Only very basic integer sorting algorithms such as radix sorting and merge sorting, are used. The current routines for these sorts in our programs are plain and have not been optimized. All the I/O jobs in our programs are executed via the operating system (by calling functions \texttt{fread} and \texttt{fwrite} in C).

In their current implementations, \textsc{dsais}, \textsc{emsads} and \textsc{esais} do not use compression to reduce the I/O volume and disk use. To the best of our knowledge, \textsc{bwt-disk} is a representative (and may be the only up-to-date) method for using compression to improve the practical performance of sorting suffixes in the external memory. The study in [Perragina et al. 2012] showed that compressing data can make disk use smaller than the size of the input string. Both \textsc{dsais} and \textsc{emsads} perform I/Os in size-\( W \) blocks. They can naturally be improved by using compression on each I/O block. Nevertheless, due to the use of a priority queue provided by \textsc{STXXL}, adding a compression feature to I/Os in \textsc{esais} may have to be done inside \textsc{STXXL}. As we are not experts in \textsc{STXXL}, we are not sure if this can be done and leave it to be determined by interested readers.

### 5. SUMMARY

The core contribution of this paper is to introduce a new disk-friendly method in \textsc{dsa-is} for sequentially retrieving the preceding character of a sorted suffix to induce the order of the preceding suffix. There are several potential applications for this method. For example: By modifying \textsc{sa-is}, Fischer [Fischer 2011] gave an excellent algorithm-
m to induce an LCP array. As the induced sorting process in DSA-IS exactly mimics its counterpart in SA-IS, the method for inducing the LCP array can naturally be extended to the external memory model with DSA-IS. Goto and Bannai [Goto and Bannai 2013] utilized SACAK to design a space efficient linear-time algorithm for computing LZ77 factorization on constant alphabets. Kärkkäinen, Kempa and Puglisi [Kärkkäinen et al. 2014] proposed algorithms for computing the LZ77 parsing efficiently using the external memory. This suggests a possibility for extending DSA-IS for computing LZ77 factorization in the external memory. The approach for turning SA-IS into DSA-IS can be directly applied to make an external memory design for the suffix array construction algorithm in [Ko and Aluru 2005].

The algorithms eSAIS, EM-SA-DS and DSA-IS contain three designs proposed for using the induced sorting principle to sort suffixes in the external memory. For readers’ information, some comparisons between them are sketched below.

— Given the internal memory capacity $M = \Omega((nW)^{0.5})$ and the external memory capacity $E = O(n)$, the I/O complexity of each is $O(n/W)$.
— The best experimental times are achieved by DSA-IS and eSAIS. They are around 10% better than that of EM-SA-DS. Although they have similar speed, eSAIS uses around 20% more disk space than DSA-IS.
— Random access of $bkt[x[j]]$ and $sa[bkt[x[j]]]$ (described in Section 1) is avoided by dividing the suffix array into blocks in both DSA-IS and EM-SA-DS, whereas eSAIS uses a priority queue managed by the external memory library STXXL.
— Random access of $x[j]$ (described in Section 1) is avoided in different ways. The methods used in eSAIS and EM-SA-DS both divide long LMS-substrings into fixed-size substrings and move the substrings around during the inducing process. However, DSA-IS uses a technique based on dividing $x$ into blocks and constructing a separate suffix array for each block, in which the main inducing phase can be regarded as a multiway merging of the suffix arrays of the blocks.

DSA-IS is presented here to share the results of our current research on designing efficient suffix sorting algorithms. We are approaching a favorable position for the development of a distributed solution for building a big suffix array or suffix tree and their compressed alternatives. We are currently investigating efficient methods for extending our external memory algorithms to be distributed, so as to further scale the problem size by running the algorithms on a distributed system consisting of many computing nodes.

ACKNOWLEDGMENTS

The authors wish to thank the reviewers who have given constructive and insightful suggestions for improving the presentation of this paper.

REFERENCES


Induced Sorting Suffixes in External Memory


