Inducing Suffix and LCP Arrays in External Memory*

Timo Bingmann†, Johannes Fischer‡, and Vitaly Osipov§

KIT, Institute of Theoretical Informatics, 76131 Karlsruhe, Germany
{timo.bingmann,johannes.fischer,osipov}@kit.edu

Abstract

We consider full text index construction in external memory (EM). Our first contribution is an inducing algorithm for suffix arrays in external memory, which utilizes an efficient EM priority queue and runs in sorting complexity. Practical tests show that this algorithm outperforms the previous best EM suffix sorter [Dementiev et al., JEA 2008] by a factor of about two in time and I/O-volume. Our second contribution is to augment the first algorithm to also construct the array of longest common prefixes (LCPs). This yields the first EM construction algorithm for LCP arrays. The overhead in time and I/O volume for this extended algorithm over plain suffix array construction is roughly two. Our algorithms scale far beyond problem sizes previously considered in the literature (text size of 80 GiB using only 4 GiB of RAM in our experiments).

1 Introduction

Suffix arrays [25, 16] are among the most popular data structures for full text indexing. They list all suffixes of a static text in lexicographically ascending order. This not only allows to efficiently locate arbitrary patterns in unstructured texts (like DNA, East Asian languages, etc.) in time proportional to the *pattern* length (as opposed to *text* length), but also fast phrase searches (e.g., "to be or not to be") if the suffix array is built over the phrase beginnings only [11].

The first and most important step in using suffix arrays is the efficient construction of the index (also called "*suffix sorting*"), the term "efficient" encompassing both time and space. Construction of suffix arrays in internal memory is a particularly well studied problem.

Nevertheless it is remarkable that until 2009, the text indexing community was confronted with the dilemma that there were theoretically fast algorithms for constructing suffix arrays in internal memory (linear-time for integer alphabets) that were rather slow in practice [1], while other superlinear algorithms existed that outperformed the linear ones on all realistic instances, in terms of both time and space [28, 33, 26, etc.]. In particular, the extremely elegant *difference cover algorithm* (DC3 for short) by Kärkkäinen et al. [21], is reported to be 3–4 times slower than the best superlinear solutions, even with very careful implementations [31].

This situation changed when in 2009 Nong et al. [29] presented another extremely elegant linear time algorithm called SAIS *that was also fast in practice*, which was based on the induced sorting principle [17]. Despite being almost in-place and faster than (or almost as fast as) all previous algorithms on all *practical* inputs, its worst-case guarantees also imply that it has a similar behavior on *all* inputs, while for all engineered superlinear algorithms, like those mentioned in the preceding paragraph, there exist "bad" inputs where the running time shoots up by several order of magnitudes.

Nonetheless, the simplicity of the DC3 algorithm (mostly sorting and scanning) enables straightforward adaptation to more advanced models of computation (PRAM, EM, distributed, etc.), and usually leads to asymptotically optimal algorithms in those models. In fact, there is a fast EM implementation of DC3 [6] that outperformed all other external suffix sorters in practice at the time of its writing. Other external implementations of DC3 (and its variant DC7) confirmed those results [8].

In many applications (e.g., for fast string matching), the suffix array needs to be augmented with the *longest common prefix array* (LCP array), which holds the lengths of longest common prefixes of lexicographically consecutive suffixes. In internal memory, the LCP array can be constructed sufficiently fast [22, 27, 18, 15]. In the EM model, the DC3 suffix sorter can be augmented to also construct the LCP array within sorting complexity. However, we are not aware of any previous implementation of this approach. Another purely theoretical solution is to use the EM suffix *tree*

*A version of this paper was presented at ALENEX'13 [5]
†Supported by DFG SPP 1307.
‡Supported by the German Research Foundation (DFG).
§Partially supported by EU Project No. 248481 (PEPPHER) ICT-2009.3.6

algorithm [9] for constructing LCP arrays and derive the LCP array by an EM Euler tour over the tree. This approach seems even less suitable for an efficient implementation. There are only a couple of semi-external construction algorithms [15, 18, 34], where "semi-external" means that they only need *some* arrays in main memory, while other parts can be scanned.

We point out that a truly external LCP array construction algorithm is the only missing piece for a fast practical EM suffix *tree* construction, because, as Barsky et al. [4, p. 986] say in their survey on EM suffix *trees*: "The conversion of a suffix array into a suffix tree turned out to be disk-friendly, since reads of the suffix array and writes of the suffix tree can be performed sequentially. However, the suffix array needs to be augmented with the LCP information in order to be converted into a suffix tree." They also comment on the possibility of adapting external DC3 to LCP arrays: "It is currently not clear how efficient the presented algorithm for the LCP computation would be in a practical implementation." And finally they say: "It may be only one step that divides us from a scalable solution for constructing suffix trees on disk for inputs of any type and size. Once this is done, a whole world of new possibilities will be opened, especially in the field of biological sequence analysis." The present paper closes this gap, as outlined in the following section "Our Contributions."

1.1 Our Contributions and Outline.

Motivated by the superior performance of the SAIS algorithm over other suffix array construction algorithms in internal memory, in this paper we investigate how the induced sorting principle can be exploited in the EM model. We have two goals in mind: (1) engineer an EM suffix sorting algorithm that outperforms the currently best one [6] while keeping it within sorting complexity, and (2) implement the *first* external memory LCP array construction algorithm that is faster than a DC3-based approach. Both of our algorithms are the first EM algorithms based on the induced sorting principle [29]. Thus, we make the first comparative study of suffix sorting in EM that includes algorithms based on the induced sorting principle, since all previous studies [6, 4] were conducted before the advent of SAIS.

In section 3, we show that SAIS is suitable for the EM model by reformulating the original algorithm such that it uses only scanning, sorting, merging, and *priority queues*. The former three operations are certainly doable in EM, and there are also EM priority queues achieving sorting lower bounds in theory [2] and practice [32, 7]. We make some careful implementation decisions in order to keep the I/O-volume low. As a result, our new algorithm, called eSAIS, is about two times faster than the EM-implementation of DC3 [6]. The I/O volume is reduced by a similar factor. In section 4 we engineer the first fully EM algorithm for LCP array construction. It is 3–4 times faster than our own implementation of LCP construction using DC3 (recall there was no such implementation before). The increase in both time and I/O volume of eSAIS with LCP array construction compared to pure suffix array construction is only around two.

Our experimental results are given in section 5. There, we apply our algorithms on problem sizes previously never considered in the literature. In sum, all experiments reported in this paper took 34 computing days and 200 TiB I/O volume. At the extreme end, we could build the suffix-array for an 80 GiB XML dump of the English Wikipedia in 2.5 µsec per character using only 4 GiB of main memory, with a total of about 18 TiB of generated I/O-volume. Such massive results have never been reported before.

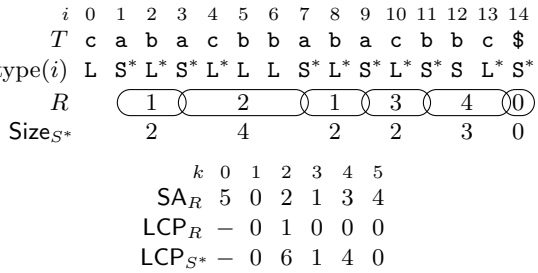
1.2 Further Related Work.

General-purpose EM string sorting routines have been described by Arge et al. [3]. There are also practical EM methods for constructing related text indexes like the Burrows-Wheeler transform [12]. A recent external memory algorithm for suffix and LCP array [24] focuses on "generalized" suffix array construction, where the input is a set of independent small strings, which each fits into internal memory.

2 Preliminaries

Let $[0,n] := \{0,\ldots,n\}$ and $[0,n) := \{0,\ldots,n-1\}$ be ranges of integers. Given a string $T = [t_0 \ldots t_{n-1}]$ of n characters drawn from a totally ordered alphabet Σ, we call the substring $T_i := [t_i \ldots t_{n-1}]$ the i-th suffix of T. For a simpler exposition, we assume that t_{n-1} is a unique character '$' that is lexicographically smallest, although our implementation does not rely on such a sentinel character. The *suffix array* SA_T of T is the permutation of the integers $[0,n)$, such that $T_{\mathsf{SA}_T[i-1]} < T_{\mathsf{SA}_T[i]}$ (lexicographic order is always intended when comparing strings by "<"). We denote the inverse permutation of SA_T by ISA_T. The companion array LCP_T is defined as $\mathsf{LCP}_T[i] := \textsc{lcp}_T(\mathsf{SA}_T[i-1], \mathsf{SA}_T[i])$, where $\mathsf{LCP}_T[0]$ remains undefined and $\textsc{lcp}_T(i,j)$ is the length of the longest common prefix (LCP) of the suffixes T_i and T_j.

The algorithms in this paper are written in a tuple pseudo-code language, which mixes Pascal-like control flow with array manipulation and mathematical set notation. This enables powerful expressions like $A := [(i^2 \bmod 7, i) \mid i \in [0,5)]$, which sets A to



$[t_i \dots t_j]$	i	type(i)	rep(j)	LCP_N
[$]	14	S	0	–
[aba]	1	S	0	0
[aba]	7	S	0	3
[acbba]	3	S	0	1
[acb]	9	S	1	4
[bbc$]	11	S	0	0

Figure 1: Example of the structures before and after the recursive call of the induced sorting algorithm. Left: the top part shows the text, the classification of suffixes and the reduced string R on which the algorithm is run recursively. The resulting suffix and LCP arrays for R are shown in the lower part (SA_R and LCP_R). Whereas the former has a direct correspondence to the S*-suffixes in T, the latter needs to be expanded to LCP_{S^*} to account for the different alphabets in T and R (see section 4.2). Right: additional information needed to expand LCP_R to LCP_{S^*}, consisting of the sorted S*-substrings and associated information. The last column LCP_N shows the LCPs of lexicographically consecutive S*-substrings.

be the array of pairs $[\,(0,0),(1,1),(4,2),(2,3),(2,4)\,]$. The individual operations in the tuple pseudo-code are implementable in EM using appropriate algorithms: for example $(i,j) \in A$ resembles a scan over array A, and $A' := \texttt{Sort}(A)$ calls an EM sorting algorithm, which by default sorts tuples lexicographically.

2.1 Induced Sorting Toolkit. Following previous work [29], we classify all suffixes into two *types*: S and L. For suffix T_i the type(i) is S if $T_i < T_{i+1}$, and L otherwise. Suffix T_{n-1} is fixed as type S. Furthermore, we distinguish the "left-most" occurrences of either type as S* and L*; more precisely, T_i is S* if T_i is S-type and T_{i-1} is L-type. Symmetrically, T_i is L*-type if T_i is L-type and T_{i-1} is S-type. The last suffix $T_{n-1} = [\$]$ is always S*, while the first suffix is never S* nor L*. Sometimes we also say the character t_i is of type(i).

Using these classifications, one can identify subsequences within the suffix array. The range of suffixes starting with the same character c is called the c-bucket, which itself is composed of a sequence of L-suffixes followed by S-suffixes. We also define the *repetition count* for a suffix T_i as $\text{rep}(i) := \max_{k \in \mathbb{N}_0}\{t_i = t_{i+1} = \dots = t_{i+k}\}$; then the L/S subbuckets can further be decomposed into ranges of equal repetition counts, which we call *repetition buckets*.

The principle behind *induced sorting* is to deduce the lexicographic order of unsorted suffixes from a set of already ordered suffixes. Many fast suffix sorting algorithms incorporate this principle in one way or another [31]. They are built on the following *inducing lemma* [23]:

LEMMA 2.1. *If the lexicographic order of all* S*-*suffixes is known, then the lexicographic order of all* L*-*suffixes can be induced iteratively smallest to largest.*

Proof. We start with $\mathcal{L} := S^*$ as the lexicographically ordered set of S*-suffixes. Iteratively, choose the unsorted L-suffix $T_i \notin \mathcal{L}$ that, among all unsorted L-suffixes, has smallest first character t_i and smallest rank of suffix T_{i+1} within $\mathcal{L}$, such that T_{i+1} is already in $\mathcal{L}$. From these properties, $T_i < T_j$ for all $T_j \in \mathcal{L} \setminus \{T_i\}$ follows due to the transitive ordering of L-suffix chains, and T_i can be inserted into $\mathcal{L}$ as the next larger L-suffix. This procedure ultimately sorts all L-suffixes, because each has an S*-suffix to its right.

Analogously, the order of all S-suffixes can be induced iteratively largest to smallest, if the relative order of all L*-suffixes is known. Therefore, it remains to find the relative order of S*-suffixes.

For each S*-suffix T_i, we define the S**-substring** $[t_i, \dots, t_j]$, where T_j is the next S*-suffix in the string. The last S*-suffix [$] is fixed to be a sentinel S*-substring by itself. We call the last character t_j of each S*-substring the *overlapping character*, since it is also the first character of the next S*-substring. S*-substrings are ordered lexicographically, with each component compared first by character and then by type, L-characters being smaller than S-characters in case of ties. This partial order allows one to apply *lexicographic naming* to S*-substrings [29]. By representing each S*-substring by its lexicographic name in the super-alphabet Σ^*, one can efficiently solve the problem of finding the relative order of S*-suffixes by *recursively* suffix sorting the reduced string R of lexicographic names of S*-substrings. Figure 1 shows an example contain all the arrays used in this paper.

3 Induced Suffix Sorting in External Memory

We now design an EM algorithm based on the induced sorting principle that runs in sorting complexity and

Algorithm 1: eSAIS description in tuple pseudo-code

```
1  eSAIS(T = [t_0 ... t_{n-1}]) begin
2      Scan T back-to-front, create [ (s*_k) | k ∈ [0,K) ] for K S*-suffixes, and sort S*-substrings:
           P := Sort_{S*}[ ([t_i ... t_j], i, type(j)) | (i,j) = (s*_k, s*_{k+1}), k ∈ [0,K) ]   // with s*_K := n-1
3      N = [ (n_k, i) ] := Lexname_{S*}(P)        // choose lexnames n_k ∈ [0,K) for S*-substrings
4      R := [ n_k | (n_k, i) ∈ Sort(N by second component) ]        // sort lexnames back to string order
5      if the lexnames in N are not unique then
6          SA_R := eSAIS(R)                                        // recursion with |R| ≤ |T|/2
7          ISA_R := [ r_k | (k, r_k) ∈ Sort[ (SA_R[k], k) | k ∈ [0,K) ] ]        // invert permutation
8      else                                  // (Sort sorts lexicographically unless stated otherwise.)
9          ISA_R := R                                              // ISA_R has been generated directly
10     S* := [ (t_j, S, ISA_R[k], [t_{j-1} ... t_i], j) | (i,j) = (s*_{k-1}, s*_k), k ∈ [0,K) ]   // with s*_{-1} := 0
11     ρ_L := 0,   Q_L := CreatePQ(S* by (t_i, y, r, [t_{i-1} ... t_{i-ℓ}], i))
12     while (t_i, y, r, [t_{i-1} ... t_{i-ℓ}], i) = Q_L.extractMin() do        // induce from next S*- or L-suffix
13         if y = L then A_L.append((t_i, i))                       // save i as next L-type in SA
14         if t_{i-1} ≥ t_i then Q_L.insert(t_{i-1}, L, ρ_L++, [t_{i-2} ... t_{i-ℓ}], i-1)        // T_{i-1} is L-type?
15         else L*.append( (t_i, L, ρ_L++, [t_{i-1} ... t_{i-ℓ}], i) )        // T_{i-1} is S-type
16     Repeat lines 11–15 and construct A_S from L* array with inverted PQ order and ρ_S--.
17     return [ i | (t,i) ∈ Merge((t_i,i) ∈ A_L and (t_j,j) ∈ A_S.reverse() by first component) ]
```

has a lower constant factor than DC3 [6]. The basis for this algorithm is an efficient EM priority-queue (PQ) [7], as suggested by the proof of lemma 2.1. Since it is derived from RAM-based SAIS, we call our new algorithm eSAIS (*External Suffix Array construction by Induced Sorting*). We first comment on details of the pseudo-code shown as algorithm 1, which is a simplified variant of eSAIS. For a solution to complications that arise due to long S*-substrings, please refer to our full paper [5], as these cases do not occur very often.

Let R denote the reduced string consisting of lexicographic names of S*-suffixes. The objective of lines 2–9 is to create the inverse suffix array ISA_R, containing the ranks of all S*-suffixes in T. In line 2, the input is scanned back-to-front, and the type of each suffix i is determined from t_i, t_{i+1}, and type$(i+1)$. Thereby, S*-suffixes are identified, and we assume there are K S*-suffixes with $K-1$ S*-substrings between them, plus the sentinel S*-substring. For each S*-substring, the scan creates one tuple. These tuples are then sorted as described at the end of section 2.1. After sorting, in line 3 the S*-substring tuples are lexicographically named with respect to the S*-substring ordering, and the output tuple array N is naturally ordered by names $n_k \in [0, K)$. The names must be sorted back to string order in line 4. This yields the reduced string R, wherein each character represents one S*-substring. If the lexicographic names are unique, the lexicographic ranks of S*-substrings are simply the names in R (lines 8–9). Otherwise the ranks are calculated recursively by calling eSAIS and inverting SA_R (lines 5–7).

With ISA_R containing the ranks of S*-suffixes, we apply lemma 2.1 in lines 10–15. The PQ contains quintuples $(t_i, y, r, [t_{i-1}, \ldots, t_{i-\ell}], i)$ with (t_i, y, r) being the sort key, which is composed of character t_i, indicator $y = \text{type}(i)$ with L < S and relative rank r of suffix T_{i+1}. To efficiently implement lemma 2.1, instead of checking *all* unsorted L-suffixes, we design the PQ to create the relative order of S*- and L-suffixes as described in the proof. Extraction from the PQ always yields the smallest unsorted L-suffix, or, if all L-suffixes within a c-bucket are sorted, the smallest S*-suffix i with unsorted preceding L-suffix at position $i-1$ (hence $t_{i-1} > c$). Thus diverging slightly from the proof, the PQ only contains L-suffixes T_i where T_{i+1} is already ordered, plus all S*-suffixes where T_{i-1} has not been ordered; so at any time the PQ contains at most K items. In line 11, the PQ is initialized with the array S^*, which is built in line 10 by reading the input back-to-front again, re-identifying S*-suffixes and merging with ISA_R to get the rank for each tuple. Notice that the characters of S*-substrings are saved in *reverse* order. The while loop in lines 12–15 then repeatedly removes the minimum item and assigns it the next relative rank as enumerated by ρ_L; this is the *inducing* process. If the extracted tuple represents an L-suffix, the suffix position i is saved in A_L as the next L-suffix in the t_i-bucket (line 13). Extracted S*-suffixes do not have an output. If the preceding suffix T_{i-1} is L-type, then we shorten the tuple by one character to represent this suffix, and reinsert

the tuple with its relative rank (line 14). However, if the preceding suffix T_{i-1} is S-type, then the suffix T_i is L*-type, and it must be saved for the inducing of S-suffixes (line 15). When the PQ is empty, all L-suffixes are sorted in A_L, and L^* contains all L*-suffixes ranked by their lexicographic order.

With the array L^* the while loop is repeated to sort all S-suffixes (line 16). This process is symmetric with the PQ order being reversed and using ρ_S-- instead of incrementing. If $t_{i-1} > t_i$ occurs, the tuple can be dropped, because there is no need to recreate the array S^* (as all L-suffixes are already sorted). When both A_L and A_S are computed, the suffix array can be constructed by merging together the L- and S-subsequences bucket-wise (line 17). A_S has to be reversed first, because the S-suffix order is generated largest to smallest. Note that in this formulation the alphabet Σ is only used for comparison.

4 Inducing the LCP Array in External Memory

In this section we describe the first practical algorithm that calculates the LCP array in external memory. The general method of integrating LCP construction into SAIS has already been described [13]; here, we adapt it to the EM model and have to deal with issues that did not arise in the RAM implementation [13] because the latter was not implemented recursively. From the recursion, we can assume that the LCP array LCP_R of the reduced string R is calculated together with SA_R, while in the base case with unique lexicographic names LCP_R is simply filled with zeros. Calculation of the LCP array LCP_T of the original text is done in two phases: first LCP_R is expanded to the array LCP_{S^*} containing the LCPs of lexicographically consecutive S*-suffixes of T, and from these the LCP values of all other suffixes are induced by solving semi-dynamic range minimum queries (RMQs) in EM.

4.1 Expanding the Recursive LCP Array.

Given the recursively calculated LCP array LCP_R, we first show how to calculate $\mathsf{LCP}_{S^*}[k] := \mathrm{LCP}_T(s^*_{\mathsf{SA}_R[k-1]}, s^*_{\mathsf{SA}_R[k]})$, which is the maximum number of equal characters (in T, not in R!) starting at two lexicographically consecutive S*-suffixes. See again figure 1, which also gives an example of all concepts presented in this section.

There are two main issues to deal with: firstly, a reduced character in R is composed of several characters in T. Apart from the obvious need for *scaling* the values in LCP_R by the lengths of the corresponding S*-substrings, we note that even *different* characters in R can have a common prefix in T and thus contribute to the total LCP. For example, in figure 1 the first two S*-substrings [aba] and [acbba] both start with an 'a', although they are different characters in R. The second issue is that lexicographically consecutive S*-suffixes can have LCPs encompassing more than one S*-substring in one suffix, but not in the other. For example, the S*-suffix T_3 = [acbbabacbbc$] and T_9 = [acbbc$] have an LCP of 4 that spans two S*-substrings of the latter suffix.

To handle both issues, additional information must be precalculated during the S*-substring splitting and lexicographic naming steps in lines 2–3 of algorithm 1. During splitting in line 2, the S*-substring tuples must be amended with the repetition count of the overlapping character, $P := \mathtt{Sort}_{S^*}[\,([t_i, \ldots, t_j], i, \mathrm{type}(j), \mathrm{rep}(j)) \mid (i,j) = (s^*_k, s^*_{k+1}),\, k \in [0, K)\,]$, which must also influence the sorting and naming of S*-substrings. Furthermore, we store the length of each S*-substring, minus the one overlapping character, in an array called $\mathsf{Size}_{S^*} := [\, s^*_{k+1} - s^*_k \mid k \in [0, K)\,]$ in string order. Lastly, during lexicographic naming, we compute the LCPs of lexicographically consecutive S*-sub*strings* in an array LCP_N, and later use (static) RMQs over LCP_N to find the common characters of arbitrary S*-suffixes.

The final formula for computing LCP_{S^*} is given by

$$\mathsf{LCP}_{S^*}[k] = \sum_{i=\mathsf{SA}_R[k]}^{\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]-1} \mathsf{Size}_{S^*}[i] + \mathrm{RMQ}_{\mathsf{LCP}_N}(\ell[k], r[k]) \tag{4.1}$$

$$\text{with} \quad \ell[k] = \mathsf{ISA}_R\big[\mathsf{SA}_R[k-1] + \mathsf{LCP}_R[k]\big] + 1$$

$$\text{and} \quad r[k] = \mathsf{ISA}_R\big[\mathsf{SA}_R[k] + \mathsf{LCP}_R[k]\big]\,,$$

where the first part sums over the sizes of the common lexicographic names of consecutive S*-suffixes, and the RMQ delivers the LCP of the following unequal pair, as explained above. If $\mathsf{LCP}_R[k] = 0$, then the whole expression reduces to $\mathsf{LCP}_N[k]$, as one would except.

We omit a fine detail about LCP_N here: in (e)SAIS, components of S*-substrings are compared first by character and then by type. For LCP construction, however, we are interested only in the common characters. This complication is dealt with in detail in our full paper [5].

4.2 Calculating LCP_{S^*} in External Memory.

Having established how LCP_{S^*} is in principle calculable, we now discuss how to implement the algorithm in the EM model. According to equation 4.1, two subproblems must be solved efficiently in external memory: range sums over Size_{S^*}, and range minimum queries over LCP_N. The first is solved by preparing query tuples for the sum boundaries and then performing a prefix-sum scan on Size_{S^*}. In more detail, from two consecutive entries, prepare two range sum query tuples $(\mathsf{SA}_R[k] - 1, k)$, $(\mathsf{SA}_R[k] + \mathsf{LCP}_R[k] - 1, k)$, sort these by first component, and perform a prefix-sum scan on Size_{S^*}, which

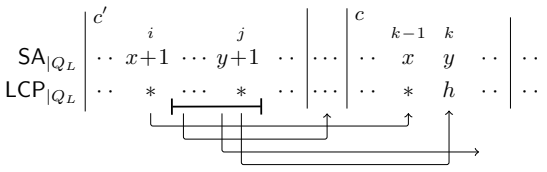


Figure 2: General scheme of the inducing step. When inducing k, the LCP value $h = \text{RMQ}_{\mathsf{LCP}_T}(i+1, j) + 1$ can be derived using an RMQ between the previous and current relative ranks of the induce sources.

delivers $\sum_{k=0}^{\mathsf{SA}_R[k]-1}$ and $\sum_{k=0}^{\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]-1}$, from which the range sum is easily calculated.

For the static range minimum queries in LCP_N, we follow a common RAM-technique [14]: we precompute $\mathcal{O}(n)$ potential subqueries by a scan of LCP_N, and store them on disk. The actual queries are divided into three subqueries, sorted, and merged with the precomputed queries (first by left, then by right query end). A final sort by query IDs brings the answers to subqueries back together. This technique was already sketched in the DC3 algorithm [21].

4.3 Computing LCPs by Finding Minima. We now explain how to construct the LCP array LCP_T of the input string T, given the LCP-values of $\mathtt{S}^*$-suffixes in T in the array LCP_{S^*}, as explained above. The general idea is to follow the inducing mechanism as explained in section 3 and induce the LCP-values along with the suffix array values [13].

First look at the inducing of L-suffixes (lines 11–15 in algorithm 1). For what follows, we imagine an array $\mathsf{SA}_{|Q_L}$ consisting of the suffix array values of suffixes that are extracted from the priority queue Q_L in line 12 of algorithm 1 (last element i of the quintuple), in the order as they are extracted (hence $\mathsf{SA}_{|Q_L}$ consists of the fifth components of $\mathtt{S}^*$, plus the second components of A_L). Likewise, we define the array $\mathsf{LCP}_{|Q_L}$ consisting of the corresponding LCP array values. Hence, the aim is to augment the while loop in lines 12–15 of algorithm 1 to also compute $\mathsf{LCP}_{|Q_L}$. The LCPs of $\mathtt{S}^*$-suffixes are exactly the array LCP_{S^*}, as computed above. We next show how to compute the entries in $\mathsf{LCP}_{|Q_L}$ for the L-suffixes.

Suppose that line 13 is just about to append (t_y, y) to the array $\mathsf{SA}_{|Q_L}$, right next to a tuple (t_x, x) for some $T_x < T_y$ with $t_x = t_y$. The goal is to determine h, the LCP of suffixes T_x and T_y. See also figure 2, which shows the situation in terms of the sequences $\mathsf{SA}_{|Q_L}$ and $\mathsf{LCP}_{|Q_L}$. The suffixes that caused the inducing of T_x and T_y are T_{x+1} and T_{y+1}, respectively, and due to lemma 2.1 those two latter suffixes are lexicographically smaller than suffix T_y (hence also smaller than T_x). Now observe that the suffixes T_x and T_y are exactly the suffixes T_{x+1} and T_{y+1} with the new character $t_x = t_y$ prepended. Hence, the LCP of T_x and T_y is exactly one more than the LCP of T_{x+1} and T_{y+1}.

If $t_{x+1} \neq t_{y+1}$, then the LCP is $h = 1$. Otherwise, due to the lexicographic ordering of the suffixes, the LCP of T_{x+1} and T_{y+1} can be obtained by taking the minimum of all $\mathsf{LCP}_{|Q_L}$-values between the positions of those suffixes. The LCPs of all those suffixes are already known either from LCPs of $\mathtt{S}^*$-suffixes or by induction from L-suffixes. Hence, $h := \text{RMQ}_{\mathsf{LCP}_{|Q_L}}(i+1, j) + 1$ is the true LCP-value of T_x and T_y when outputting (t_x, x) in line 13 of algorithm 1, where i and j are the positions of T_{x+1} and T_{y+1} in the partial suffix array. These positions i and j are available directly from the PQ: they are the relative ranks 'r' in the preceding and current quintuple. There remains one exception for the LCP of the last L-suffix and the first S-suffix within a bucket, however, this case is easy to handle [13] using the repetition counts of those suffixes.

The RMQs delivering the LCP values are created in batch during inducing and answered afterwards, forming the LCP array. But notice that they are *interdependent*! This implies that RMQ problem we are faced with is in fact a dynamic problem. For our external memory solution to this problem please refer to our full paper [5].

Finally, we note that we have also implemented a completely in-memory version of RMQs that relies on the fact that only the right-to-left minima (looking left from the current position) are candidates for the minima. Except for pathological inputs there are only $\mathcal{O}(M)$ such right-to-left minima, because the minimum at each bucket boundary is zero. Therefore they all fit in RAM and can be searched in a binary manner or using more involved heuristics.

5 Experimental Evaluation

We implemented the eSAIS algorithm with integrated LCP construction in C++ using the external memory library Stxxl [7]. This library provides efficient external memory sorting and a priority queue that is modeled after the design for cached memory [32]. Note that in Stxxl all I/O operations bypass the operating system cache; therefore the experimental results are not influenced by system cache behavior. Our implementation and selected input files are available from http://tbingmann.de/2012/esais/.

Before describing the experiments, we highlight some details of the implementation. Most notably, Stxxl does not support variable length structures, nor are we aware of a library with PQ that does. Therefore, in the implementation the tuples in the PQ and the

associated arrays are of fixed length, and superfluous I/O transfer volume occurs. All results of the algorithms were verified using a suffix array checker [6, Sect. 8] and a semi-external version of Kasai's LCP algorithm [22] (when possible). We designed the implementation to use an implicit sentinel instead of '$,' so that input containing zero bytes can be suffix sorted as well. Since our goal was to sort massive inputs, the implementation can use different data types for array positions: usual 32-bit integers and a special 40-bit data type stored in five bytes. The input data type is also variable, we only experimented with usual 8-bit inputs, but the recursive levels work internally with the 32/40-bit data type. When sorting ASCII strings in memory, an efficient in-place radix sort [19] is used. Strings of larger data types are sorted in RAM using `gcc-4.4` STL's version of introsort.

We chose a wide variety of large inputs, both artificial and from real-world applications:

Wikipedia is an XML dump of the most recent version of all pages in the English Wikipedia, which was obtained from `http://dumps.wikimedia.org/`; our dump is dated `enwiki-20120601`.

Gutenberg is a concatenation of all ASCII text documents from `http://www.gutenberg.org/robot/harvest` as available in September 2012. The Gutenberg data contains a version of the human genome as a sub-string.

Human Genome consists of all DNA files from the UCSC human genome assembly "hg19" downloadable from `http://genome.ucsc.edu/`. The files were normalized to upper-case and stripped of all characters but $\{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}, \mathtt{N}\}$. Note that this input contains very long sequences of unknown `N` placeholders, which influences the LCPs.

Pi are the decimals of π, written as ASCII digits and starting with "3.1415."

Skyline is an artificial string for which eSAIS has maximum recursion depth. To achieve this, the string's suffixes must have type sequence $\mathtt{LSLS}\ldots\mathtt{LS}$ at each level of recursion.

The input Skyline is generated depending on the experiment size, all other inputs are cut to size. The inputs are available from the same URL as our implementation's source code.

Our main experimental **platform A** was a cluster computer, with one node exclusively allocated when running a test instance. The nodes have an Intel Xeon X5355 processor clocked with 2.66 GHz and 4 MiB of level 2 cache. In all tests only one core of the processor is used. Each node has 850 GiB of available disk space striped with RAID 0 across four local disks of size 250 GiB; the rest is reserved by the system. We limited the main memory usage of the algorithms to 1 GiB of RAM, and used a block size of 1 MiB. The block size was optimized in preliminary experiments.

Due to the limited local disk space in the cluster computer, we chose to run some additional, larger experiments on **platform B**: an Intel Xeon X5550 processor clocked with 2.66 GHz and 8 MiB of level 2 cache. The main memory usage was limited to 4 GiB RAM, we kept the block size at 1 MiB and up to seven local SATA disk with 1 TB of local space were available.

Programs on both platforms were compiled using `g++` 4.4.6 with `-O3` and native architecture optimization.

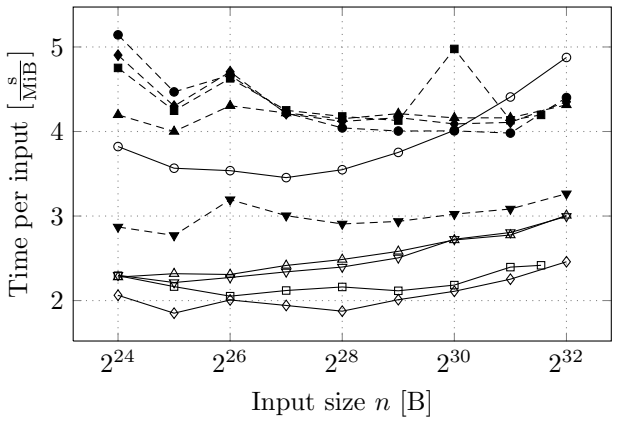
5.1 Plain Suffix Array Construction. As noted in the introduction, the previously fastest EM suffix sorter is DC3 [6]. We adapted and optimized the original source code, which is already implemented using STXXL, to our current setup and larger data types. An implementation of DC7 exists that is reported to be about 20% faster in the special case of human DNA [34], but we did not include it in our experiments.

Figure 3 shows the construction time and I/O volume of eSAIS and DC3 on platform A using 32-bit keys. The two algorithms eSAIS (open bullets) and DC3 (filled bullets) were run on prefixes $T[0, 2^k)$ of all five inputs, with only Skyline being generated specifically for each size. In total these plots took 3.2 computing days and over 16.8 TiB of I/O volume, which is why only one run was performed for each of the 90 test instances.

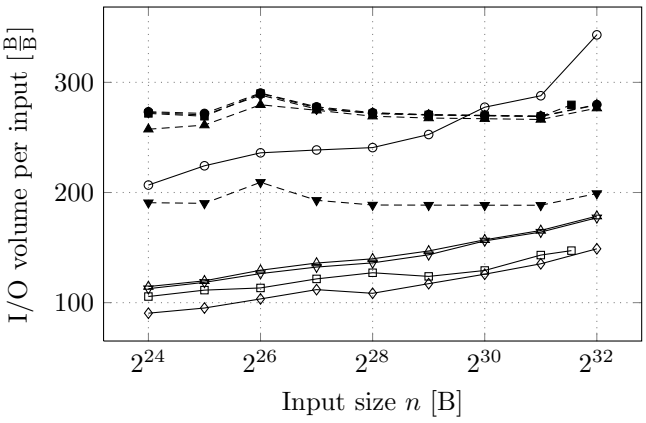
For all real-world inputs eSAIS's construction time is about half of DC3's. The I/O volume required by eSAIS is also only about 60% of the volume of DC3. The two artificial inputs exhibit the extreme results they were designed to provoke: Pi is random input with short LCPs, which is an easy case for DC3. Nevertheless, eSAIS is still faster, but not twice as fast. The results from eSAIS's worst-case Skyline show another extreme: eSAIS has highest construction time on its worst input, whereas DC3 is moderately fast because Skyline can efficiently be sorted by triples. The high I/O volume of eSAIS for Skyline is due to its maximum recursion depth, reducing the string only by $\frac{1}{2}$ and filling the PQ with $\frac{n}{2}$ items on each level. The PQ implementation requires more I/O volume than sorting, because it recursively combines short runs to keep the arity of mergers in main memory small. Even though DC3 reduces by $\frac{2}{3}$, the recursion depth is limited by $\log_3 n$ and sorting is more straightforward.

Besides the basic eSAIS algorithm, we also implemented a variant which "discards" sequences of multiple unique names from the reduced string prior to recursion [6, 30]. However, we discovered that this optimization

(a) Construction Time



(b) I/O Volume



(c) Input Characteristics

Plot	Input	$\lvert T\rvert$	$\lvert\Sigma\rvert$
—△—	Wikipedia XML	79 479 MiB	213
—◇—	Gutenberg Text	22 975 MiB	256
—□—	Human Genome	2 992 MiB	5
—▽—	Decimals of pi	∞	11
—○—	Skyline (worst-case)	∞	$\lfloor \log_2 n \rfloor$

—○— eSAIS -●- DC3
(all inputs)

(d) Average LCP

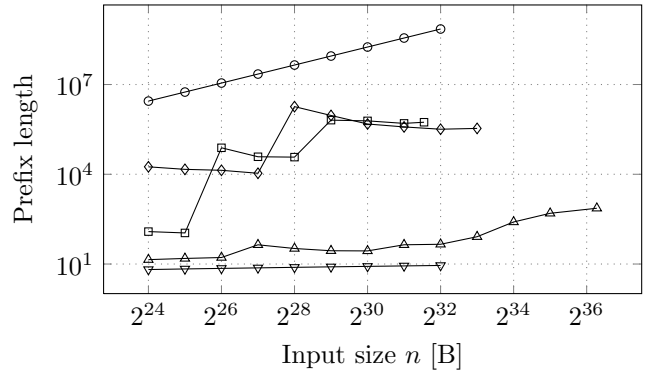


Figure 3: The top two plots show (a) construction time and (b) I/O volume of eSAIS (open bullets) and DC3 (filled bullets) on experimental platform A. The table (c) shows selected characteristics of the input strings. Subfigure (d) contains the average LCP of increasing 2^k slices of the inputs, calculated using eSAIS-LCP.

has much smaller effect in eSAIS than in other suffix sorters (see figure 4 (a)-(d)). This is probably due to the induced sorting algorithm already adapting very efficiently to the input string's characteristics.

5.2 Suffix and LCP Array Construction. We implemented two variants of LCP construction: one solving RMQs in EM (LCPext), and the other entirely in RAM (LCPint). The EM solution saves RMQs to disk during the inducing process, and constructs the LCP array from these queries after the SA was completed. Contrarily, the RAM solution precalculates the LCP for each induced position from an in-memory structure and saves the LCP in the PQ. Thus the LCP array is constructed at the same time as the SA (when extracting from the PQ). The size of the in-memory RMQ structure is related to the maximum LCP and the number of different inducing targets within one bucket, and grows up to 300 MiB for the Human Genome. The in-memory RMQ construction also requires the preceding character t_{i-1} to be available when processing the while loop, a restriction that requires an overlap of two characters in continuation tuples and thus leads to a larger I/O volume.

Since no EM variant of DC3 with LCP construction in STXXL is available, we extended the original implementation to also calculate the LCP array recursively, as suggested in [20] and documented by our student [10]. These steps are similar to those needed in eSAIS-LCP (see section 4.2), however, DC3-LCP generally requires two batched random lookups and two generally unpredictable RMQs per output value. In eSAIS-LCP on the other hand, the lexicographic names encompass variable length substrings, thus requiring the prefix-sum, followed by the same batched random access and an RMQ on LCP_N. But due to the structure of the induc-

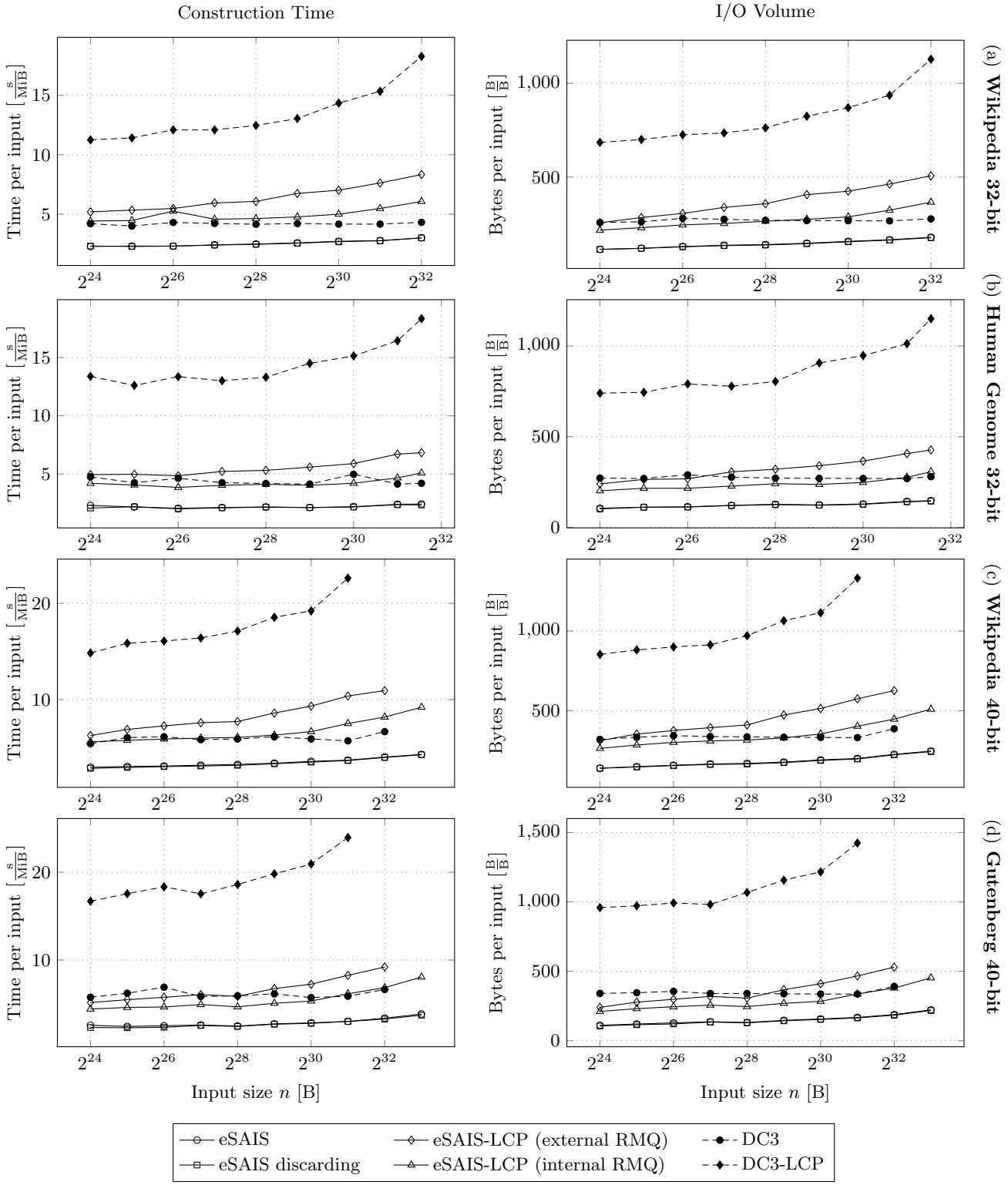


Figure 4: Subfigures (a)-(d) show construction time and I/O volume of all six implementations run on platform A for three different inputs. Subfigures (a)-(b) use 32-bit positions, while (c)-(d) runs with 40-bit. On the right hand side, $\frac{\text{B}}{\text{B}}$ indicates I/O volume in bytes per input byte.

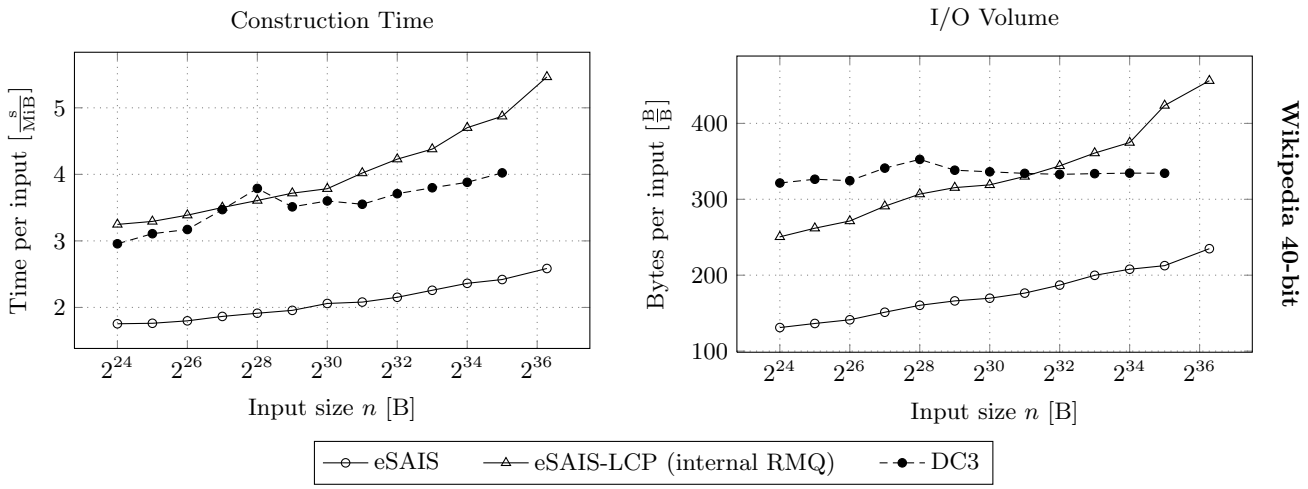


Figure 5: Measured construction time and I/O volume of three implementations is shown for the largest test instance Wikipedia run on platform B using 40-bit positions

ing process, fewer operations are required after calculating LCP_{S^*} and the RMQ ranges are "local" to the currently induced bucket.

Figure 4 (a)-(d) shows the results of all six variants of the algorithms on the real-world inputs run on platform A. We observe that eSAIS-LCP internal or external are the first viable methods to calculate suffix array and LCP array in EM; our version of DC3-LCP finishes in justifiable time only for very small instances. On all real-world inputs the construction time of eSAIS-LCP is never more than twice the time of DC3 *without* LCP construction. As expected, in-memory RMQs are consistently faster than EM-RMQs and also require fewer I/Os, even though the PQ tuples are larger.

To exhibit experiments with building large suffix arrays, we configured the algorithms to use 40-bit positions on platform A. Figure 4 (c)-(d) show results for the Wikipedia and Gutenberg input only up to 2^{33}, because larger instances require more local disk space than available at the node of the cluster computer. On average over all tests instances of Wikipedia, calculation using 40-bit positions take about 33% more construction time and the expected 25% more I/O volume.

The size of suffix arrays that can be built on platform A was limited by the local disk space; we therefore determined the maximum disk allocation required. Table 1 shows the average maximum disk allocation measured empirically over our test inputs for 32-bit and 40-bit offset data types.

On platform B we had the necessary 4 TiB disk space required to process the full Wikipedia instance, and these results are shown in figure 5. The maximum size of the in-memory RMQ structure was only about 12 MiB. Sorting of the whole Wikipedia input with eSAIS took 2.4 days and 18 TiB I/O volume, and with eSAIS with LCP construction (internal memory RMQs) took 5.0 days and 35 TiB I/O volume.

	eSAIS	-LCPint	-LCPext	DC3	-LCP
32-bit	$25n$	$44n$	$52n$	$46n$	$88n$
40-bit	$28n$	$54n$	$63n$	$58n$	$109n$

Table 1: Maximum disk allocation in bytes required by the algorithms, averaged and rounded over all our inputs

6 Conclusions and Future Work

We presented a better external memory suffix sorter that can also construct the LCP array. Although our implementations are already very practical, we point out some optimizations that could yield an even better performance in the future. Because eSAIS is largely compute bound, a more efficient internal memory priority queue implementation, e.g. a radix heap, may improve suffix array construction time significantly. Obviously, for real-world applications one should stop sorting in external memory when the reduced string can be suffix sorted internally. This is currently not implemented. Another fact that could lead to significantly better performance is that any reinsertion into the PQ is always after the last tuple of the current repetition bucket. Thus the PQ's main-memory merge buffer could be bypassed in many cases. As a whole, the potential of further sequential speed improvements by optimization of eSAIS is higher than for DC3.

References

[1] Antonitio, P. J. Ryan, W. F. Smyth, A. Turpin, and X. Yu. New suffix array algorithms — linear but not fast? In *Proc. Fifteenth Australasian Workshop Combinatorial Algorithms (AWOCA)*, pages 148–156, 2004.

[2] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proc. STOC*, pages 540–548. ACM Press, 1997.

[4] M. Barsky, U. Stege, and A. Thomo. A survey of practical algorithms for suffix tree construction in external memory. *Softw. Pract. Exper.*, 40(11):965–988, 2010.

[5] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proc. ALENEX*, pages 88–102. SIAM, 2013.

[6] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM J. Exp. Algorithmics*, 12:Article No. 3.4, 2008.

[7] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[8] A. Döring, D. Weese, T. Rausch, and K. Reinert. SeqAn — an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9:11, 2008.

[9] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[10] D. Feist. External batched range minimum queries and LCP construction. Bachelor Thesis at Karlsruhe Institute of Technology, April 2013.

[11] P. Ferragina and J. Fischer. Suffix arrays on words. In *Proc. CPM*, volume 4580 of *LNCS*, pages 328–339. Springer, 2007.

[12] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[13] J. Fischer. Inducing the LCP-array. In *Proc. WADS*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011.

[14] J. Fischer and V. Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[15] S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proc. ALENEX*, pages 25–34. SIAM Press, 2011.

[16] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 3, pages 66–82. Prentice-Hall, 1992.

[17] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. SPIRE/CRIWG*, pages 81–88. IEEE Press, 1999.

[18] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proc. CPM*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

[19] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *Proc. SPIRE*, volume 5280 of *LNCS*, pages 3–14. Springer, 2009.

[20] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. *Proc. ICALP*, 2719:943–955, 2003.

[21] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):1–19, 2006.

[22] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[23] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2–4):143–156, 2005.

[24] F. A. Louza, G. P. Telles, and C. D. D. A. Ciferri. External memory generalized suffix and LCP arrays construction. In *Proc. CPM*, volume 7922 of *LNCS*, pages 201–210. Springer, 2013.

[25] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[26] M. A. Maniscalco and S. J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM J. Exp. Algorithmics*, 12:Article no. 1.2, 2008.

[27] G. Manzini. Two space saving tricks for linear time lcp array computation. In *Proc. Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

[28] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[29] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[30] S. J. Puglisi, W. F. Smyth, and A. Turpin. The performance of linear time suffix sorting algorithms. In *Proc. Data Compression Conf. (DCC)*, pages 358–367. IEEE Computer Society, 2005.

[31] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[32] P. Sanders. Fast priority queues for cached memorys. *ACM J. Exp. Algorithmics*, 5:Article No. 7, 2000.

[33] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Softw. Pract. Exper.*, 37(3):309–329, 2007.

[34] D. Weese. Entwurf und Implementierung eines generischen Substring-Index. Master's thesis, Humboldt University Berlin, May 2006.