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 text index construction in external memory (EM). Our first contribution is an inducing algorithm for suffix arrays in external memory. Practical tests show that this outperforms the previous best EM suffix sorter [Dementiev et al., ALENEX 2005] 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 [16, 24] are among the most popular data structures for full text indexing. They list all suffixes of a static text in lexicographically increasing 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 ("*suffix sorting*"), the term "efficient" encompassing both time and space. Until recently, the text indexing community was confronted with the dilemma that there were theoretically fast algorithms for constructing suffix arrays (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 [25,26,31]. In particular, the extremely elegant *difference cover algorithm* (DC3 for short) by Kärkkäinen et al. [21], which has quickly become a showcase string algorithm and is now being taught in many computer science classes around the world, is reported to be 3–4 times slower than the best superlinear solutions, even with very careful implementations [29].

This situation changed when in 2009 Nong et al. [27, we cite more recent journal versions whenever possible] presented another extremely elegant linear time algorithm called SAIS *that was also fast in practice* (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 [25,26,31, etc.] 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 optimal algorithms in those models. In fact, there is a fast EM implementation of DC3 [7] that outperformed all other external suffix sorters in practice at the time of its writing. Other external implementations of DC3 (or its variant DC7) confirmed those results [9].

In many applications (e.g., for fast string matching), the suffix array needs to be augmented with the *longest common prefix array* (LCP array for short), which holds the lengths of longest common prefixes of lexicographically consecutive suffixes. In internal memory, the LCP array can be constructed sufficiently fast. Indeed, the currently fastest algorithm [13] also uses the induced sorting framework on which SAIS is based. 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* algorithm [10] for constructing LCP arrays and derive the LCP array by an EM Euler tour over the tree. This approach

*Supported by DFG SPP 1307.
†Supported by the German Research Foundation (DFG).
‡Partially supported by EU Project No. 248481 (PEPPHER) ICT-2009.3.6

seems even less suitable for an efficient implementation. There are only a couple of semi-external construction algorithms [15, 18, 33], where "semi-external" means that they only need *some* arrays in RAM, 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.

Motivated by the superior performance of the SAIS algorithm over other suffix array construction algorithms in internal memory, in this paper we investigate if the induced sorting principle can be exploited also in the EM model. We have two goals in mind: (1) engineer an EM suffix sorting (hence also BWT!) algorithm that outperforms the currently best one [7] 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 based on the induced sorting principle [27]. Thus, we make the first comparative study of suffix arrays in EM that includes the induced sorting principle, since all previous studies [4, 7] were conducted before the advent of SAIS. In § 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 both in theory [2] and in practice [8, 30]. 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 [7]. The I/O volume is reduced by a similar factor. We then proceed and 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 algorithms scale far beyond problem sizes previously 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 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 paper [5] describes an EM LCP array construction algorithm for the specific case of short DNA-reads (which is, due to the quadratic dependency on the length of the longest read, not suitable for arbitrary strings). A completely different research topic not pursued here is how to use an external suffix array to efficiently *answer queries*; see e.g. [32].

2 Preliminaries

Let $[0, n] := \{0, \ldots, n\}$ and $[0, n) := \{0, \ldots, n-1\}$ be ranges of integers, and $\mathbb{1}_{cond} \in \{0, 1\}$ be a boolean variable indicating the truth of condition *cond*.

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 $\mathsf{LCP}_T[i] := \text{LCP}_T(\mathsf{SA}_T[i-1], \mathsf{SA}_T[i])$, where $\mathsf{LCP}_T[0]$ remains undefined and $\text{LCP}_T(i, j)$ is the length of the longest common prefix (LCP) of the suffixes T_i and T_j.

2.1 Induced Sorting Toolkit.

Following previous work [27], 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 $\mathtt{S}^*$ and $\mathtt{L}^*$; more precisely, T_i is $\mathtt{S}^*$ if T_i is S-type and T_{i-1} is L-type. Symmetrically, T_i is $\mathtt{L}^*$-type if T_i is L-type and T_{i-1} is S-type. The last suffix $T_{n-1} = [\$]$ is always $\mathtt{S}^*$, while the first suffix is never $\mathtt{S}^*$ nor $\mathtt{L}^*$. 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 $\mathrm{rep}(i) := \max_{k\in\mathbb{N}_0}\{t_i = t_{i+1} = \cdots = t_{i+k}\}$; 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 [29]. They are built on the following *inducing lemma* [23]:

LEMMA 2.1. *If the lexicographic order of all* $\mathtt{S}^*$*-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 $\mathtt{S}^*$-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 $\mathtt{S}^*$-suffix to its right.

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

For each $\mathtt{S}^*$-suffix T_i, we define the $\mathtt{S}^*$*-substring* $[t_i, \ldots, t_j]$, where T_j is the next $\mathtt{S}^*$-suffix in the string. The last $\mathtt{S}^*$-suffix $[\$]$ is fixed to be a sentinel $\mathtt{S}^*$-substring by itself. We call the last character t_j of each $\mathtt{S}^*$-substring the *overlapping character*. $\mathtt{S}^*$-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 $\mathtt{S}^*$-substrings [27]. By representing each $\mathtt{S}^*$-substring by its lexicographic name in the superalphabet Σ^*, one can efficiently solve the problem of finding the relative order of $\mathtt{S}^*$-suffixes to recursively suffix sorting the reduced string of lexicographic names of $\mathtt{S}^*$-substrings.

3 Induced Suffix Sorting in External Memory

Our first goal is to design an EM algorithm based on the induced sorting principle that runs in sorting complexity and has a lower constant factor than DC3 [7]. The basis for this algorithm is an efficient EM priority-queue (PQ) [8], as suggested by the proof of lem. 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 pseudocode shown as alg. 1, which is a simplified variant of eSAIS. § 3.1 is then devoted to complications that arise due to large $\mathtt{S}^*$-substrings.

Let R denote the reduced string consisting of lexicographic names of $\mathtt{S}^*$-suffixes. The objective of lines 2–9 is to create the inverse suffix array ISA_R, containing the ranks of all $\mathtt{S}^*$-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, $\mathtt{S}^*$-suffixes are identified, and we assume there are K $\mathtt{S}^*$-suffixes with $K-1$ $\mathtt{S}^*$-substrings between them, plus the sentinel $\mathtt{S}^*$-substring. For each $\mathtt{S}^*$-substring, the scan creates one tuple. These tuples are then sorted as described at the end of § 2.1 (note that the type of each character inside the tuple can be deduced from the characters and the type of the overlapping character). After sorting, in line 3 the $\mathtt{S}^*$-substring tuples are lexicographically named with respect to the $\mathtt{S}^*$-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 $\mathtt{S}^*$-substring. If the lexicographic names are unique, the lexicographic ranks of $\mathtt{S}^*$-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 $\mathtt{S}^*$-suffixes, we apply lem. 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 $y = \mathrm{type}(i)$ with $\mathtt{L} < \mathtt{S}$ and relative rank r of suffix T_{i+1}. To efficiently implement lem. 2.1, instead of checking *all* unsorted L-suffixes, we design the PQ to create the relative order of $\mathtt{S}^*$- 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 $\mathtt{S}^*$-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 $\mathtt{S}^*$-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

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) ]
```

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. See fig. 4 for an example of this process.

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.

3.1 Splitting Large Tuples.

After the detailed description of alg. 1, we must point out two issues that occur in the EM setting. While S*-substrings are usually very short, at least three characters long and on average four, in pathological cases they can encompass nearly the whole string. Thus in line 2–3 of alg. 1, the tuples would grow larger than an I/O block B, and one would have to resort to long string sorting [3]. More seriously, in the special case of [$] being the only S*-suffix, the while-loop in lines 12–15 inserts $\frac{n(n+1)}{2}$ characters, which leads to quadratic I/O volume. Both issues are due to long S*-substrings, but we will deal with them differently.

Long string sorting in EM can be dealt with using lexicographic naming and doubling [3, Sect. 4]. However, instead of explicitly sorting long strings, we integrate the doubling procedure into the suffix sorting recursion and ultimately only need to sort short strings in line 2 of alg. 1. This is done by dividing the S*-substrings into *split substrings* of length at most B, starting at the *beginning*, and lexicographically naming them along with all other substrings. Thereby, a long S*-substring is represented by a sequence of lexicographic names in the reduced string. The corresponding split tuples are formed in the same way as S*-substring tuples in P, they also overlap by one character, except that this character need not be S*-type. After the recursive call, long S*-substrings are correctly ordered among all other S*-substring due to suffix sorting, and split tuples can easily be discarded in line 10 as they do not

Algorithm 2: Inducing step with S*-substrings split by D_0 and D, replacing lines 10–15 of alg. 1

```
1  D := { s*_k − D_0 − ν·D | ν ∈ ℕ, s*_k − D_0 − ν·D > s*_{k−1}, k ∈ [0, K) }        // split positions, with s*_{−1} = 0
2  S* := Sort[ (t_j, ISA_R[k], [t_{j−1} … t_i], j, 𝟙_{i∈D}) | j = s*_k, i = max(s*_{k−1}, j − D_0), k ∈ [0, K) ]
3  L := Sort[ (t_j, rep(j), j, [t_{j−1} … t_i], 𝟙_{i∈D}) | j ∈ D, i = max(s*_{k−1}, j − D), t_j is L-type ]
4  S := Sort[ (t_j, rep(j), j, [t_{j−1} … t_i], 𝟙_{i∈D}) | j ∈ D, i = max(s*_{k−1}, j − D), t_j is S-type ]
5  ρ_L := 0,  a := ⊥,  r_a = 0,  S* := Stack(S*),  Q_L := CreatePQ(∅ by (t_i, r, [t_{i−1} … t_{i−ℓ}], i, c))
6  while Q_L.NotEmpty() or S*.NotEmpty() do
7      while Q_L.Empty() or t < Q_L.TopChar() with (t, r, [t_{i−1} … t_{i−ℓ}], i, c) = S*.Top() do
8          Q_L.insert(t_{i−1}, ρ_L++, [t_{i−2} … t_{i−ℓ}], i − 1, c),  S*.Pop()        // induce from S*-suffixes
9      a′ := a,  a := Q_L.TopChar(),  r_a := (r_a + 1)𝟙_{a′=a},  m := ρ_L,  M := ∅    // next a-repetition bucket
10     while Q_L.TopChar() = a and Q_L.TopRank() < m do                                // induce from L-suffixes
11         (t_i, r, [t_{i−1} … t_{i−ℓ}], i, c) = Q_L.extractMin(),  A_L.append((t_i, i))   // save i as next L-type
12         if ℓ > 0 then
13             if t_{i−1} ≥ t_i then Q_L.insert(t_{i−1}, ρ_L++, [t_{i−2} … t_{i−ℓ}], i − 1, c)   // T_{i−1} is L-type
14             else L*.append( (t_i, ρ_L++, [t_{i−1} … t_{i−ℓ}], i, c) )                 // T_{i−1} is S-type
15         else if ℓ = 0 and c = 1 then M.append(i, ρ_L++, )                            // need continuation?
16     foreach Merge([ (a, r_a, i, r) | (i, r) ∈ Sort(M) ] with (a, r_a, i, [t_{i−1}, …, t_{i−ℓ}], c) ∈ L) do
17         if t_{i−1} ≥ t_i then Q_L.insert(t_{i−1}, r, [t_{i−2} … t_{i−ℓ}], i − 1, c)   // T_{i−1} is L-type
18         else L*.append( (a, r, [t_{i−1} … t_{i−ℓ}], i, c) )                           // T_{i−1} is S-type
```

correspond to any S*-suffix. The *d-critical* version of SAIS [27, Sect. 4] is a similar approach.

The second issue arises due to repeated re-insertions of payload characters into the PQ in line 14, possibly incurring quadratic I/O volume. This again is handled by splitting the S*-substrings, now starting at the *end*, into chunks of size D_0 or D ($D_0 \geq D$ indicating when to split at all, and $D \geq 1$ being the actual split length). Lines 10–15 of alg. 1 have to be replaced by alg. 2, which we will describe in the following. Let $\mathcal{D}$ be the set of splitting positions, counting first D_0 and then D characters backwards starting at each S*-suffix until the preceding S*-suffix is met. As before, for each S*-substring a tuple is stored in the S^* array, except that only the initial D_0 payload characters are copied. We call these items *seed tuples*. If an S*-substring consists of more than D_0 characters, a *continuation tuple* is stored in one of the two new arrays L or S in lines 3–4, depending on the type of its overlapping character. This overlapping character t_i will later be used together with its *repetition count* $\mathrm{rep}(i)$ to efficiently match continuation tuples with preceding tuples (see § 2.1 for the definition of repetition counts); $\mathrm{rep}(i)$ is easily calculated while reading the text back-to-front. Along with both seed and continuation tuples we save a flag $\mathbb{1}_{i\in\mathcal{D}}$ marking whether a continuation exists.

Differing from alg. 1, in line 5 the PQ is initialized as empty and S^* will be processed as a stack. This modification separates the while loop into inducing from S*-suffixes in lines 7–8 and inducing from L-suffixes in lines 10–15. The two induction sources are alternated between, with precedence depending on their top character: $Q_L.\texttt{TopChar}() = t_i$ with $(t_i, r, \tau, i, c) = Q_L.\texttt{Top}()$. Since L-suffixes are smaller than S*-suffixes if they start with the same character, the while loop in 7–8 may only induce from S*-suffixes with the first character being smaller than $Q_L.\texttt{TopChar}()$; otherwise, the while loop in 10–15 has precedence. When line 9 is reached, the loop in 10–15 extracts all suffixes from the PQ starting with a, after which the S^* stack must be checked again. In lines 11–14 the extracted tuple is handled as in alg. 1, however, when there is no preceding character t_{i-1} in the tuple and the continuation flag c is set, the tuple *underruns* and the matching continuation must be found. For each underrun tuple, the required position i and its assigned rank ρ_L is saved in the buffer M, which will be sorted and merged with the L array in line 16. Matching of the continuation tuple can be postponed up to the smallest rank at which a continued tuple may be reinserted into the PQ. This earliest rank is $m = \rho_L$, as set in line 9, because any reinsertion will have $r \geq \rho_L$, and thus the while loop 10–15 extracts exactly the r_a-th repetition bucket of a. Because continuation tuples must only be matched exactly once per repetition bucket, the continuation tuples are sorted by $(t_j, \mathrm{rep}(j), j)$, whereby L can be sequentially merged with M if M is kept sorted by the first component and L scanned as a stack.

In § 4 we compute the optimal values for D_0 and D, and analyze the resulting I/O volume.

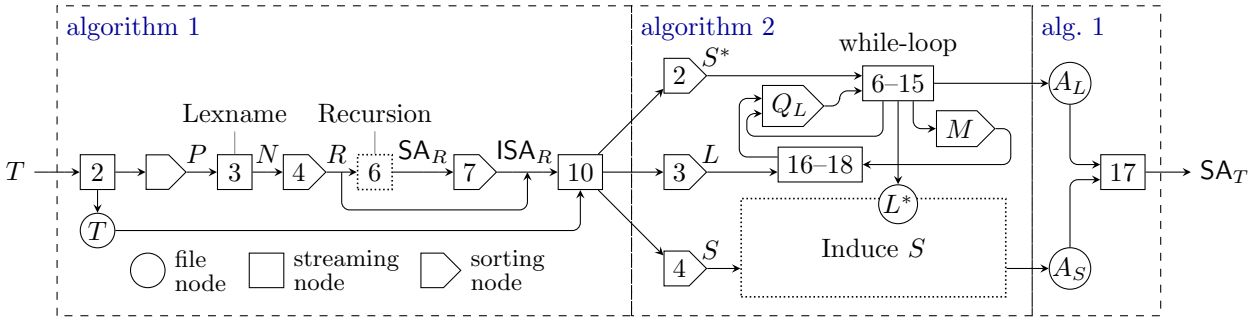


Figure 1: Data flow graph of the algorithm; numbers refer to the line numbers of alg. 1 and alg. 2, respectively. The input T is read and saved to a file (2), while creating tuples. Sorting these tuples yields P, whose entries are lexicographically named in N (3) and sorted again by string index, resulting in R (4). If names are not unique in R, the algorithm calls itself recursively (6) to calculate SA_R. The suffix array is inverted into ISA_R (7) and resulting ranks are merged with T to create seed and continuation tuples (10), which are distributed into sorters (2,3,4) in alg. 2. The main while-loop (6–15) reads from array S^* and priority-queue Q_L. Depending on the calculation, the while-loop outputs final $\mathtt{L}$-suffix order information into A_L, stores merge requests to M when tuples underrun, reinserts a shortened tuple, or it saves $\mathtt{L}^*$-tuples. Merge requests are handled by matching tuples from M and L (16–18) and reinserting into Q_L. When the while-loop for inducing $\mathtt{L}$-suffixes finishes, the process is repeated with seed tuples from $\mathtt{L}^*$ and continuation tuples from S, yielding the final $\mathtt{S}$-suffix order values in A_S. The output suffix array is constructed by merging A_L and A_S (17).

4 I/O Analysis of eSAIS with Split Tuples

We now analyze the overall I/O performance of our algorithm and find the best splitting parameters D_0 and D, both under practical assumptions. We will focus on calculating the I/O volume processed by `Sort` in lines 2–4 and 16, and by the PQs.

For simplicity, we assume that there is only one elemental data type, disregarding the fact that characters can be smaller than indices, for instance. Thus a tuple is composed of multiple elements of equal size. We write $\textsc{Sort}(n)$ or $\textsc{Scan}(n)$ as the number of I/Os needed to sort or scan an array of n elements. For our practical experiments we assume $n \leq \frac{M^2}{B}$, and thus can relate $\textsc{Sort}(n) = 2\,\textsc{Scan}(n)$, which is equivalent to saying that n elements can be sorted with one in-memory merge step. With parameters $M = 2^{30}$ (1 GiB) and $B = 2^{10}$ (1 MiB), as used in our experiments, up to 2^{50} (1 PiB) elements can be sorted under this assumption. Furthermore, we also assume that the PQ has amortized I/O complexity $\textsc{Sort}(n)$ for sorting n elements, an assumption that is supported by preliminary experiments.

In the analysis we denote the length of $\mathtt{S}^*$-substrings *excluding* the overlapping character, thus the sum of their lengths is the string length. For further simplicity, we assume that line 15 of alg. 2 always stores continuation requests in M, and unmatched requests are later discarded. Thus our analysis can ignore the boolean continuation variables.

For a broader view of the algorithm, we abstracted alg. 1 (including alg. 2) into a pipelined data flow graph in fig. 1.

Lemma 4.1. *To minimize I/O cost alg. 2 should use $D = 3$ and $D_0 = 8$ for splitting $\mathtt{S}^*$-strings.*

Proof. We first focus on the number of elements sorted and scanned by the algorithm for one $\mathtt{S}^*$-substring of long length $\ell = kD$ for $k \in \mathbb{N}_1$ when splitting by period D and set $D_0 := D$. In this proof we count amortized costs $\textsc{Sort}(1)$ per element sorted and $\textsc{Scan}(1)$ per element scanned. This is possible, as all $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings are processed by the algorithm sequentially.

For one $\mathtt{S}^*$-substring the algorithm incurs $\textsc{Sort}(D+3)$ for sorting $\mathtt{S}^*$ (line 2) and $\textsc{Sort}((\frac{\ell}{D}-1)\cdot(D+3))$ for sorting L and S (lines 3–4). In Q_L and Q_S a total of $\textsc{Sort}(\frac{\ell}{D}(\frac{1}{2}D(D+1)) + \ell \cdot 3)$ occurs due to repeated reinsertions into the PQs with decreasing lengths. The buffer M (line 16) requires at most $\textsc{Sort}((\frac{\ell}{D}-1)\cdot 2)$, while reading from L and S is already accounted for. Additionally, at most $\textsc{Scan}((D-1)+3)$ occurs when switching from Q_L to Q_S via L^*, as at least the first $\mathtt{S}$-character was removed. Overall, this is $\textsc{Sort}(\frac{\ell}{D}(\frac{1}{2}D^2 + \frac{9}{2}D + 5) - 2) + \textsc{Scan}(D+2)$, which is minimized for $D = \sqrt{10} \approx 3.16$, when assuming $\textsc{Sort} = 2\,\textsc{Scan}$. Taking $D = 3$, we get at most $\textsc{Sort}(\frac{23}{3}\ell - 3) + \textsc{Scan}(5)$ per $\mathtt{S}^*$-substring.

Next, we determine the value of D_0 (as the length at when to start splitting by D). This offset is due to the base overhead of using continuations over just reinserting into the PQ. Given an $\mathtt{S}^*$-substring of length ℓ, repeated reinsertions without continuations would incur $\textsc{Sort}(\frac{1}{2}\ell(\ell+1) + \ell \cdot 3))$. By putting this quadratic cost in relation to the one with splitting by $D = 3$, we get that at length $\ell \approx 7.7$ the cost in both approaches

is balanced. Therefore, we choose to start splitting at $D_0 = 8$.

THEOREM 4.1. *For a string of length n the I/O volume of alg. 1 is bounded by* $\text{SORT}(17n) + \text{SCAN}(9n)$*, when splitting with $D = 3$ and $D_0 = 8$ in alg. 2.*

Proof. To bound the I/O volume, we consider a string that consists of $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings of length ℓ, and determine the maximum volume over all $2 \le \ell \le n$, where $\ell = 2$ is the smallest possible length of $\mathtt{S}^*$-substrings, due to exclusion of the overlapping character. Alg. 1 needs $\text{SCAN}(2n)$ to read T twice (in lines 2 and 10) and $\text{SORT}(n + \frac{n}{\ell} \cdot 2)$ to construct P in line 2, counting the overlapping character and excluding the boolean type, which can be encoded into i. In this SORT the I/O volume of $\mathtt{Lexname}_{S^*}$ is already accounted for. Creating the reduced string R requires sorting of N, and thus $\text{SORT}(2 \cdot \frac{n}{\ell})$ I/Os. Then the suffix array of the reduced string R with $|R| \le \frac{n}{\ell}$ is computed recursively and inverted using $\text{SORT}(2 \cdot \frac{n}{\ell})$, or the names are already unique. After creating ISA_R, alg. 2 is used with the parameters derived in lem. 4.1, incurring the amortized I/O cost calculated there for all $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings. The final merging of A_L and A_S (line 17) needs $\text{SCAN}(2n)$. In sum this is

$$\begin{aligned} V(n) \le\ & \text{SCAN}(2n) + \text{SORT}(n + \tfrac{n}{\ell} \cdot 2) + \text{SORT}(\tfrac{n}{\ell} \cdot 2) \\ & + V(\tfrac{n}{\ell}) + \text{SORT}(\tfrac{n}{\ell} \cdot 2) + \text{SCAN}(2n) \\ & + \tfrac{n}{\ell} \cdot \min\{\text{SORT}(\tfrac{23}{3}\ell - 3) + \text{SCAN}(5)\,, \\ & \qquad \text{SORT}(\tfrac{1}{2}\ell(\ell+1) + \ell \cdot 3) + \text{SCAN}(\tfrac{\ell}{2})\}\,. \end{aligned}$$

Maximizing $V(n, \ell)$ for $2 \le \ell \le n$ by $\ell = 2$, we get $V(n, \ell) \le V(n, 2) \le \text{SORT}(8.5n) + \text{SCAN}(4.5n) + V(\frac{n}{2})$ and, solving the recurrence, $V(n, \ell) \le \text{SORT}(17n) + \text{SCAN}(9n)$. In § 6 a worst-case string is constructed with $\mathtt{S}^*$-substrings of length $\ell = 2$.

5 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 $\mathtt{S}^*$-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.

5.1 Expanding the Recursive LCP Array.

Given the recursively calculated LCP array LCP_R, we first show how to calculate $\mathsf{LCP}_{S^*}[k] := \text{LCP}_T(s^*_{\mathsf{SA}_R[k-1]}, s^*_{\mathsf{SA}_R[k]})$, which is the maximum number of equal characters (in T, not in R!) starting at two lexicographically consecutive $\mathtt{S}^*$-suffixes. See also fig. 2, which gives an example of all concepts presented in this section.

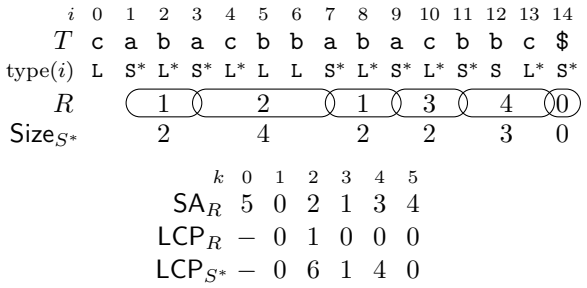
There are two main issues to deal with: firstly, a reduced character in R is composed of several characters in T. Apart from the obvious need for *scaling* the values in LCP_R by the lengths of the corresponding $\mathtt{S}^*$-substrings, we note that even *different* characters in R can have a common prefix in T and thus contribute to the total LCP. For example, in fig. 2 the first two $\mathtt{S}^*$-substrings [aba] and [acbba] both start with an 'a', although they are different characters in R. The second issue is that lexicographically consecutive $\mathtt{S}^*$-suffixes can have LCPs encompassing more than one $\mathtt{S}^*$-substring in one suffix, but not in the other. For example, the $\mathtt{S}^*$-suffix $T_3 =$ [acbbabacbbc$] and $T_9 =$ [acbbc$] have an LCP of 4 that spans two $\mathtt{S}^*$-substrings of the latter suffix.

To handle both issues, additional information must be precalculated during the $\mathtt{S}^*$-substring splitting and lexicographic naming steps in lines 2–3 of alg. 1. During splitting in line 2, the $\mathtt{S}^*$-substring tuples must be amended with the repetition count of the overlapping character, $P := \mathtt{Sort}_{S^*}[\,([t_i, \ldots, t_j], i, \text{type}(j), \text{rep}(j)) \mid (i, j) = (s^*_k, s^*_{k+1}),\ k \in [0, K)\,]$, which must also influence the sorting and naming of $\mathtt{S}^*$-substrings, as described in the next paragraph. Furthermore, we store the length of each $\mathtt{S}^*$-substring (or split string if large $\mathtt{S}^*$-substrings are split), 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 $\mathtt{S}^*$-sub*strings* in an array LCP_N, and later use (static) RMQs over LCP_N to find the common characters of arbitrary $\mathtt{S}^*$-suffixes.

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

$$\begin{aligned} \mathsf{LCP}_{S^*}[k] &= \sum_{i=\mathsf{SA}_R[k]}^{\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]-1} \mathsf{Size}_{S^*}[i] + \text{RMQ}_{\mathsf{LCP}_N}(\ell[k], r[k]) \\ \text{with}\quad \ell[k] &= \mathsf{ISA}_R\big[\mathsf{SA}_R[k-1] + \mathsf{LCP}_R[k]\big] + 1 \\ \text{and}\quad r[k] &= \mathsf{ISA}_R\big[\mathsf{SA}_R[k] + \mathsf{LCP}_R[k]\big]\,, \end{aligned} \tag{5.1}$$

where the first part sums over the sizes of the common lexicographic names of consecutive $\mathtt{S}^*$-suffixes, and the



$([t_i \dots t_j], i, \text{type}(i), \text{rep}(j))$	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 2: 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. Right: additional information needed to expand LCP_R to LCP_{S*}. The sorted array P, consisting of S*-substrings and associated information. The last column LCP_N shows the LCPs of lexicographically consecutive S*-substrings.

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 must point out 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. Thus when equal characters of different type are encountered, the number of *repetitions* of the distinguishing character that match in both S*-substrings must be added to the LCP. This is sufficient since if the same character occurs with different types, then these differing types are defined by the next differing character of each suffix, where one suffix is L and the other S, and these therefore must be different. Thus all common characters after such a position must be equal to the distinguishing character itself. In particular, this implies that we only need to look *one* S*-substring ahead.

For example, regard the penultimate row on the right side of fig. 2. Even though there are only 3 common characters in [acb] and its preceding S*-substring [acbba], there is a '4' in LCP_N because the S*-substring [acb] has a repetition count of 1.

Like the LCP calculation, the S*-substring sort order must be adapted to also encompass the repetition count of the overlapping character. As before, overlapping L characters are smaller than S characters. Of two overlapping L characters, the one with *lower* repetition count is considered as smaller. Symmetrically, of two S characters, the one with *higher* repetition count is smaller.

Having established how LCP_{S*} is in principle calculable, we now discuss how to implement the algorithm in the EM model. According to eq. (5.1), 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*}.

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 3 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].

5.2 Inducing the LCP Array. We now explain how to construct the LCP array LCP_T of the input string T, given the LCP-values of S*-suffixes in T in the array LCP_{S*}, as explained above. The general idea is to follow the inducing mechanism as explained in § 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 alg. 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 alg. 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 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 alg. 1 to also compute $\mathsf{LCP}_{|Q_L}$. The LCPs of 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_i, i) to the array $\mathsf{SA}_{|Q_L}$, right next to a tuple $(t_{i'}, i')$ for some $i' < i$ with $t_i = t_{i'}$. The goal is to determine h, the LCP of suffixes i and i' of T. See also fig. 3,

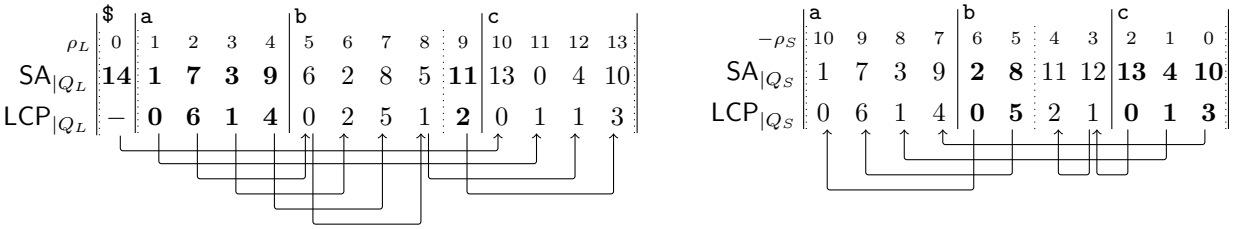


Figure 4: Example of the inducing step. Left: for the string in fig. 2, the suffix- and LCP-values of the L-suffixes (normal font) are induced from the LCPs of S*-suffixes (bold font). The variable ρ_L refers to lines 11–15 of alg. 1. The notation $\mathsf{SA}_{|Q_L}$ and $\mathsf{LCP}_{|Q_L}$ is used to denote all SA and LCP values extracted from the priority queue Q_L in line 12 of alg. 1. Right: The reverse process (line 16 of alg. 1), where the S-suffixes are induced from the L*-suffixes.

which shows the situation in terms of the (in reality nonexistent) arrays $\mathsf{SA}_{|Q_L}$ and $\mathsf{LCP}_{|Q_L}$. The suffixes that caused the inducing of i and i' are T_{i+1} and $T_{i'+1}$, respectively, and due to lem. 2.1 those two latter suffixes are lexicographically smaller than suffix $T_{i'}$ (hence also smaller than T_i). Now observe that the suffixes T_i and $T_{i'}$ are exactly the suffixes T_{i+1} and $T_{i'+1}$ with the new character t_i prepended. Hence, the LCP of T_i and $T_{i'}$ is exactly one more than the LCP of T_{i+1} and $T_{i'+1}$.

If $t_{i+1} \neq t_{i'+1}$, then the LCP is $h = 1$. Otherwise, due to the lexicographic ordering of the suffixes, the LCP of T_{i+1} and $T_{i'+1}$ 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 S*-suffixes or by induction from L-suffixes. Hence, $h := \mathrm{RMQ}_{\mathsf{LCP}_{|Q_L}}(\ell+1, r)+1$ is the true LCP-value of T_i and $T_{i'}$ when outputting (t_i, i) in line 13 of alg. 1, where ℓ and r are the positions of $i'+1$ and $i+1$ in the partial suffix array. These positions ℓ and r 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.

For example, look at the inducing of suffixes T_2 and T_8 in the left part of fig. 4. Both suffixes start with character b. The suffixes that caused the inducing are T_3 and T_9 at positions 3 and 4 of $\mathsf{SA}_{|Q_L}$, respectively, both starting with a. Their LCP is 4, which is determined by the trivial range minimum query $\mathrm{RMQ}_{\mathsf{LCP}_{|Q_L}}(4,4) = 4$. Therefore, we set $\mathsf{LCP}_{|Q_L}[7]$ to 5.

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. To solve it, we decided not to explore which of the well known EM data structures such as buffer trees [2] are suitable for solving this task within sorting complexity. Instead, we made the highly realistic assumption that the main memory size M is large enough such that $\frac{n}{M} = \mathcal{O}(M)$; or, more precisely, $n \leq C \cdot M^2$ for some small constant C (with one GiB of main memory and $C = 1/4$ as in our implementation this means we can handle problems of size $n \leq 2^{58}$, almost one Exabyte). This assumption is more lax than the one used in § 4.

Under this assumption we can split the array $\mathsf{LCP}_{|Q_L}$ into *blocks* of size $s := C \cdot M$ and keep the $\mathsf{LCP}_{|Q_L}$-values of the current block in RAM. Further, we can keep the *minima* of all $\mathcal{O}(n/M) = \mathcal{O}(M)$ previous blocks in RAM. We build succinct semi-dynamic RAM-based RMQ-structures over both arrays, as in [13, Sect. 3.2]. Then every range minimum query can be split into three subqueries: the first and last subquery being contained in a block of size s, and the middle (possibly large) subquery perfectly aligning with block boundaries on both ends. The former two subqueries are answered when the block is held in RAM, while the latter subquery is answered when the last block it contains has been processed. This takes overall $\mathcal{O}(n)$ time and $\mathcal{O}(n/B)$ I/Os.

We made some additional optimizations for cases where $\mathsf{LCP}_{|Q_L}$-values can be induced without range minimum queries. One interesting case is related to the repetition counts: consider among all L-suffixes in a c-bucket ($c \in \Sigma$) the first suffixes starting with c, cc, ccc,

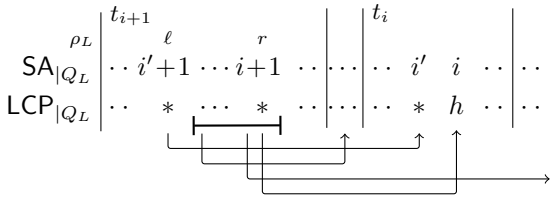


Figure 3: General scheme of the inducing step. When inducing i, the LCP value $h = \mathrm{RMQ}_{\mathsf{LCP}_{|Q_L}}(\ell+1, r)+1$ can be derived using an RMQ between the previous and current relative ranks of the induce sources, ℓ and r.

etc. Their LCP-values are 0,1,2, etc., which is exactly their repetition count. The current repetition count, however, is the known variable 'r_a' when extracting from the PQ, and thus the LCP can be set immediately without any RMQ. This optimization turned out to be very effective for highly repetitive texts.

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 i) 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 [13].

6 Experimental Evaluation

We implemented the eSAIS algorithm with integrated LCP construction in C++ using the external memory library STXXL [8]. This library provides efficient external memory sorting and a priority queue that is modeled after the design for cached memory [30]. 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. Due to fixed length structures, the results from the I/O analysis for the tuning parameter D cannot directly be used. We found that $D = D_0 = 3$ are good splitting values in practice. All results of the algorithms were verified using a suffix array checker [7, 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 large 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 `g++` STL's version of introsort. The initial sort of short strings into P was implemented using a variable length tuple sorter.

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 is obtainable 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.

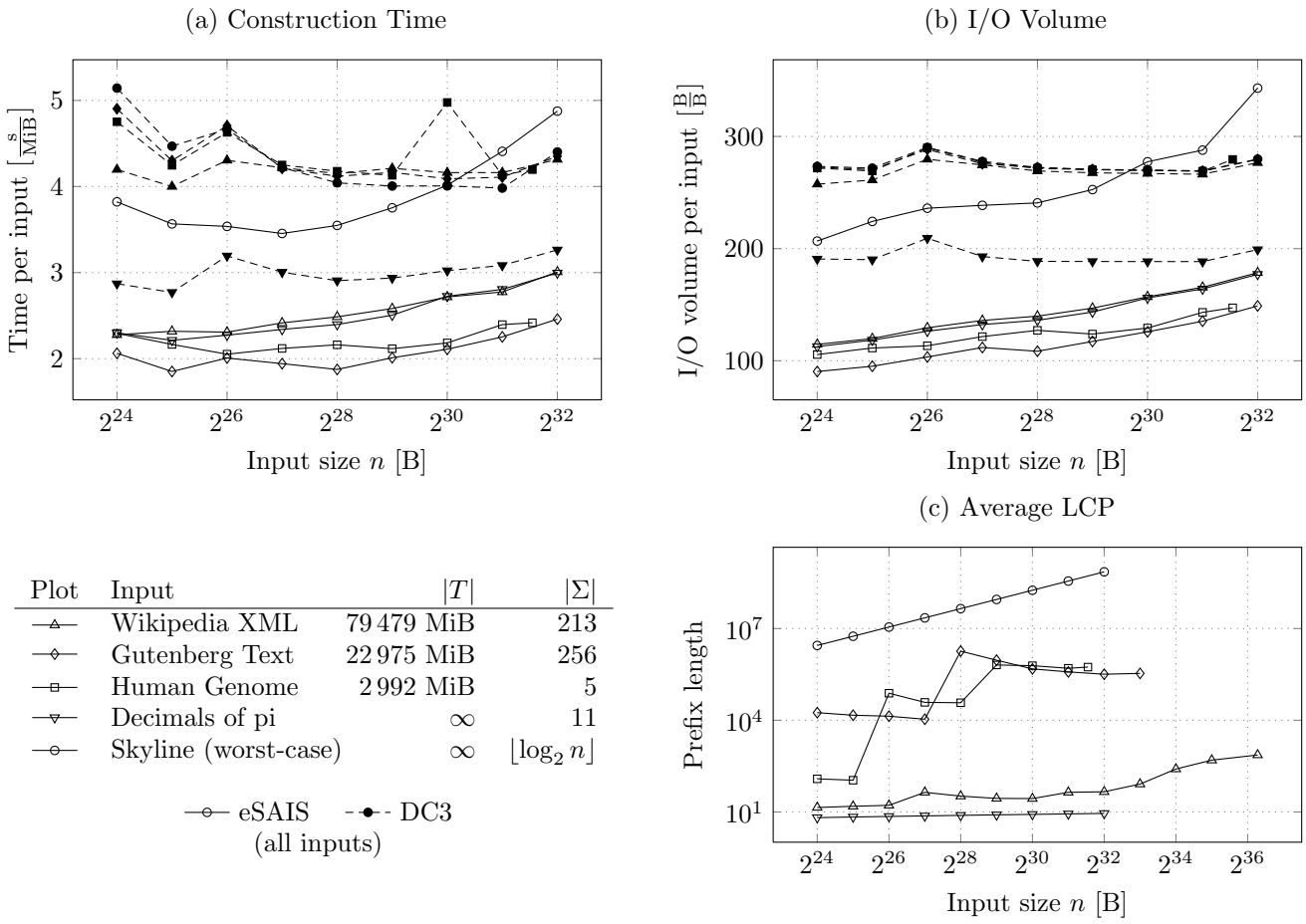
Human Genome consists of all DNA files from the UCSC human genome assembly "hg19" downloadable from http://genome.ucsc.edu/. The files were stripped of all characters but {`A`, `G`, `C`, `T`, `N`} and normalized to upper-case. Note that this input contains very long sequences of unknown `N` placeholders, 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 `LSLS`...`LS` at each level of recursion. Such a string can be constructed for a length $n = 2^p$, $p \geq 1$, using the alphabet $\Sigma = [\,\$, \sigma_1, \ldots, \sigma_p\,]$ and the grammar $\{S \to T_1\$,\ T_i \to T_{i+1}\sigma_i T_{i+1}$ for $i = 1, \ldots, p-1$ and $T_p \to \sigma_p\}$. For $p = 4$ and $\Sigma = [\,\$, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\,]$, we get `dcdbdcdadcdbdcd$`; for the test runs we replaced `$` with σ_0. The input Skyline is generated depending on the experiment size, all other inputs are cut to size.

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 six 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.



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)	∞	$\lfloor \log_2 n \rfloor$

Figure 5: The first row shows construction time and I/O volume of eSAIS (open bullets) and DC3 (filled bullets) on experimental platform A. The second row shows selected characteristics of the input strings.

6.1 Plain Suffix Array Construction. As noted in the introduction, the previously fastest EM suffix sorter is DC3 [7]. We adapted and optimized the original source code[1], 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 [33], but we did not include in our experiments.

Figure 5 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 multi-

[1] http://algo2.iti.kit.edu/dementiev/esuffix/docu/

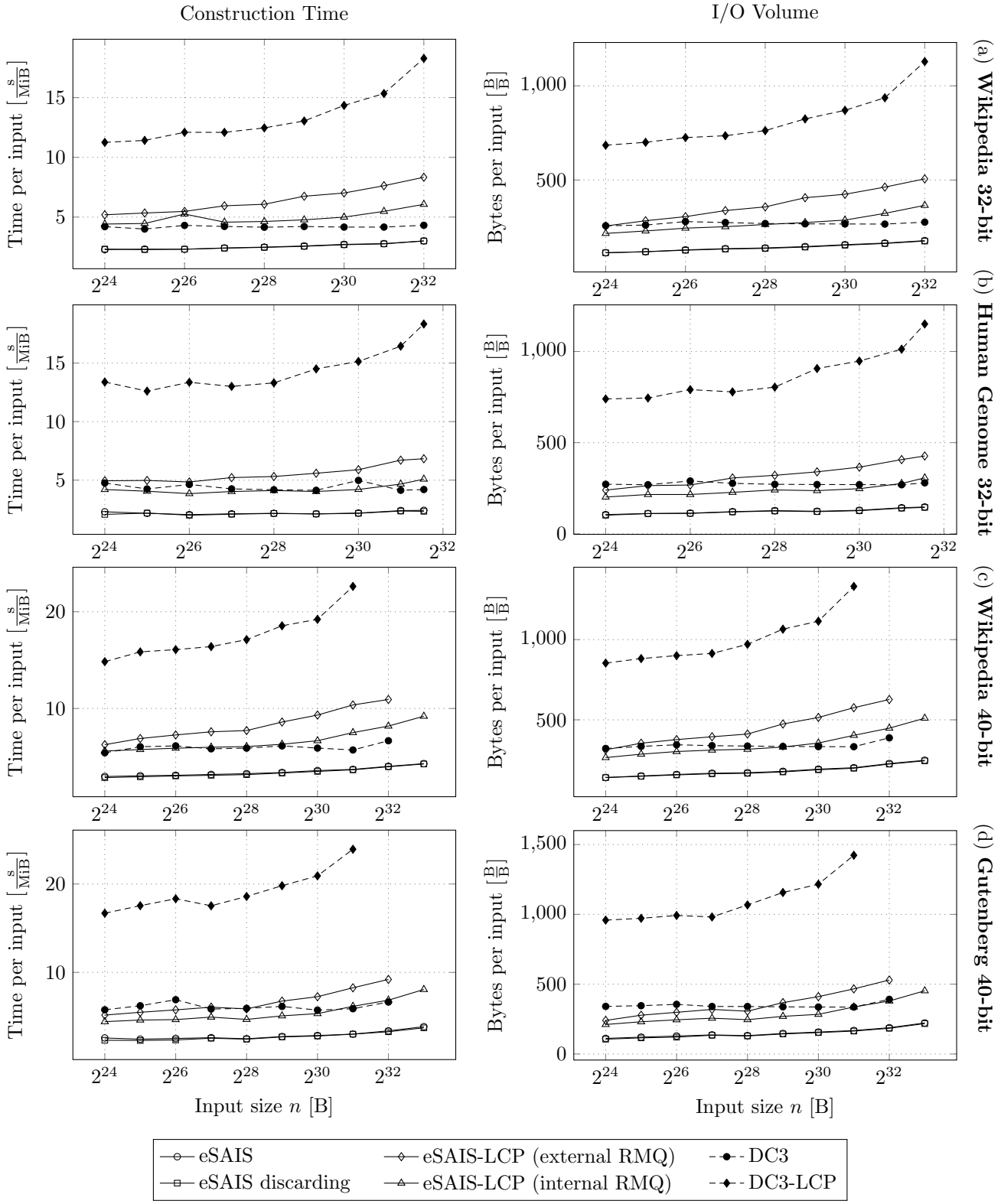


Figure 6: Subfigures (a)-(d) show construction time and I/O volume of all six implementations run on platform A for three different inputs. Subfigures (a)-(b) use 32-bit positions, while (c)-(d) runs with 40-bit. On the right hand side, $\frac{B}{B}$ indicates I/O volume in bytes per input byte.

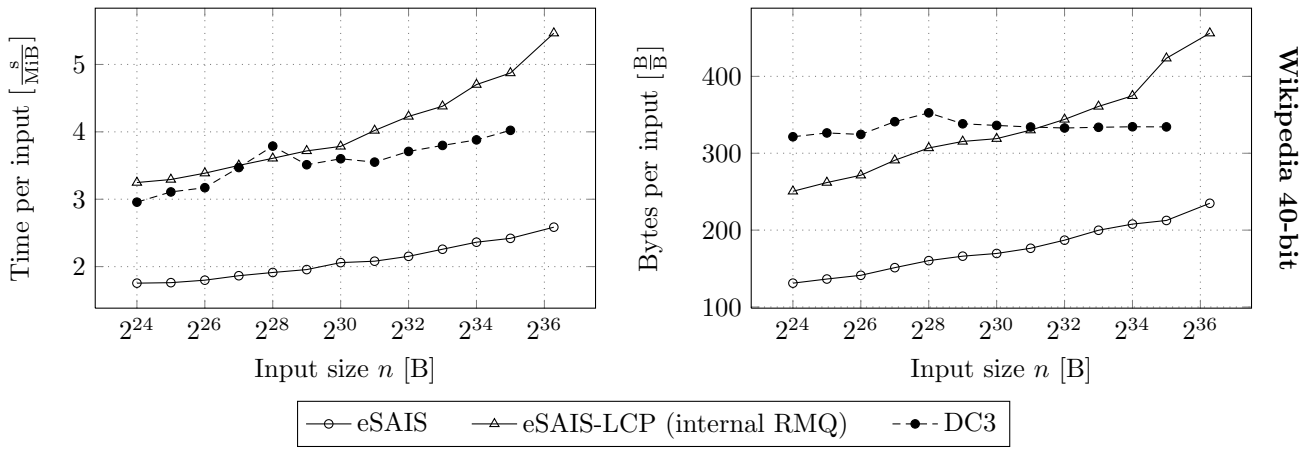


Figure 7: Measured construction time and I/O volume of three implementations is shown for the largest test instance Wikipedia run on platform B using 40-bit positions

ple unique names from the reduced string prior to recursion [7, 28]. However, we discovered that this optimization has much smaller effect in eSAIS than in other suffix sorters (see fig. 6 (a)-(d)). This is probably due to the induced sorting algorithm already adapting very efficiently to the input string's characteristics.

6.2 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 as suggested in [20].

Figure 6 (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 6 (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 fig. 7. 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

7 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. 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. Performance on inputs relying heavily on sorting (like Pi and Skyline) could also be improved by sorting $\mathtt{S}^*$-substring deeper than only three characters if they are very short. As a whole, the potential of further speed improvements by optimization of eSAIS is higher than for DC3. We also note that, the final recursive stage can also output the Burrows-Wheeler transform [6] directly from the extracted PQ tuple, instead of the suffix array. 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. Finally, it is possible to combine the two variants of eSAIS-LCP (internal and external RMQs) into one algorithm with a bounded in-memory RMQ structure, where unanswered RMQs are saved to EM and solved later.

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] M. J. Bauer, A. J. Cox, G. Rosone, and M. Sciortino. Lightweight LCP construction for next-generation sequencing datasets. In *Proc. WABI*, volume 7534 of *LNCS*, pages 326–337. Springer, 2012.

[6] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[7] 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.

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

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

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

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

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

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

[27] 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.

[28] 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.

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

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

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

[32] R. Sinha, S. J. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Proc. SIGMOD*, pages 661–672. ACM Press, 2008.

[33] D. Weese. Entwurf und Implementierung eines generischen Substring-Index. Master's thesis, Humboldt University Berlin, May 2006. http://www.seqan.de/publications/weese06.pdf.