Inducing Suffix and LCP Arrays in External Memory

Timo Bingmann*, Johannes Fischer†, and Vitaly Osipov*

*Karlsruhe Institute of Technology
†Technical University of Dortmund

Pre-print Article – First submitted April 2013, last compiled April 25, 2014

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 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 a new internal memory LCP array construction algorithm, and 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 [Manber and Myers, 1993; Gonnet et al., 1992] 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 [Ferragina and Fischer, 2007].

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. 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 [Antonitio et al., 2004], while other superlinear algorithms existed that outperformed the linear ones on all realistic instances, in terms of both time and space [Manzini and Ferragina, 2004; Schürmann and Stoye, 2007; Maniscalco and Puglisi, 2008, etc.]. In particular, the extremely elegant *difference cover algorithm* (DC3 for short) by Kärkkäinen et al. [2006], 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 [Puglisi et al., 2007].

This work is supported by the German Research Foundation (DFG) under SPP 1307, and by EU Project No. 248481 (PEPPHER) ICT-2009.3.6. Preliminary versions of this article were presented at the 12th International Symposium on Algorithms and Data Structures (WADS 2011), and at the 2013 Meeting on Algorithm Engineering & Experiments (ALENEX 2013). Author's addresses: Timo Bingmann and Vitaly Osipov: Karlsruhe Institute of Technology, Department of Informatics, 76128 Karlsruhe, Germany. Email addresses: *firstname.lastname*@kit.edu. Johannes Fischer: Technical University of Dortmund, Department of Computer Science, Chair of Algorithm Engineering (LS11), Otto-Hahn-Str. 14, 44227 Dortmund, Germany. Email address: johannes.fischer@cs.tu-dortmund.de.

This situation changed when in 2009 Nong et al. [2011] (we cite more recent journal versions whenever possible) presented another extremely elegant linear time algorithm called SAIS *that was also fast in practice*, which was based on the induced sorting principle [Itoh and Tanaka, 1999]. In addition to 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 optimal algorithms in those models. In fact, there is a fast EM implementation of DC3 [Dementiev et al., 2008a] 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 [Döring et al., 2008].

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 [Kasai et al., 2001; Manzini, 2004; Kärkkäinen et al., 2009; Gog and Ohlebusch, 2011]. 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 [Farach-Colton et al., 2000] 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 [Gog and Ohlebusch, 2011; Kärkkäinen et al., 2009; Weese, 2006], 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., 2010, 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 tries to close 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 [Dementiev et al., 2008a] 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 [Nong et al., 2011]. 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 [Dementiev et al., 2008a; Barsky et al., 2010] were conducted before the advent of SAIS. Besides outputting the suffix and LCP arrays, our algorithm can also generate the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994].

In Section 2, we first give some basic definitions and recapitulate the SAIS algorithm in internal memory. Section 3 contains the first technical contribution of this article: we show how SAIS can be augmented to also construct the LCP-array in internal memory. Then, in Section 4, 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 [Arge, 2003]. In practice, the most efficient priority queues are those of Sanders [2000] and Dementiev et al. [2008b]. 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 [Dementiev et al., 2008a]. The I/O volume is reduced by a similar factor. In Section 6 we engineer the first fully EM algorithm for LCP array construction (building on the internal memory algorithm from Section 3). 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 7. There, we apply our algorithms on very large instances. 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. In sum, all experiments reported in this paper took 34 computing days and 200 TiB I/O volume.

1.2 Further Related Work

General-purpose EM string sorting routines have been described by Arge et al. [1997]. There are also practical EM methods for constructing related text indexes like the Burrows-Wheeler transform [Ferragina et al., 2012]. A recent paper [Bauer et al., 2012] 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. [Sinha et al., 2008].

1.3 Differences to the Conference Versions

This article extends the material already presented at the 12th International Symposium on Algorithms and Data Structures [Fischer, 2011], and at the 2013 Meeting on Algorithm Engineering & Experiments [Bingmann et al., 2013]. We now give the full details and proofs for the inducing algorithm of the LCP array (Section 3). We also show more experimental results, such as details on the fill status of the used priority queues (Section 4.2) and on the recursion depth (Figure 7). We now also compare our running times with those of bwtdisk by Ferragina et al. [2012].

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 $T = [t_0 \dots t_{n-1}]$ of n characters drawn from a totally ordered alphabet Σ, we call the substring $T_i := [t_i \dots t_{n-1}]$ 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. For any array A, we write $A[\ell, r]$ to denote the sub-array of A ranging from ℓ to r.

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 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 [Nong et al., 2011], 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 if $T_i > T_{i+1}$. 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 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 call these subsequences the c-L- and c-S-*subbuckets* or just L/S-subbuckets if the character is implied by the text. 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 [Puglisi et al., 2007]. They are built on the following *inducing lemma* [Ko and Aluru, 2005]:

Lemma 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 list of S*-suffixes, and $\mathcal{U}$ as the set of unordered L-suffixes. After the following procedure, the list $\mathcal{L}$ will contain all L- and S*-suffixes in lexicographic order: Iteratively, choose the unsorted L-suffix $T_i \in \mathcal{U}$ that

(1) has smallest first character t_i, and

(2) among those with the same t_i, the one such that T_{i+1} has smallest rank within $\mathcal{L}$.

(It follows from this procedure that for the chosen suffix T_i, suffix T_{i+1} must already be in $\mathcal{L}$.) From these properties, $T_i < T_j$ follows for all L-suffixes $T_j \in \mathcal{U} \setminus \{T_i\}$, because the L-type property forms ascending chains of unsorted suffixes in $\mathcal{U}$. Due to the transitive ordering of L-suffix $>$-chains in $\mathcal{U}$ it suffices to pick the smallest of the "tails" of these chains, which

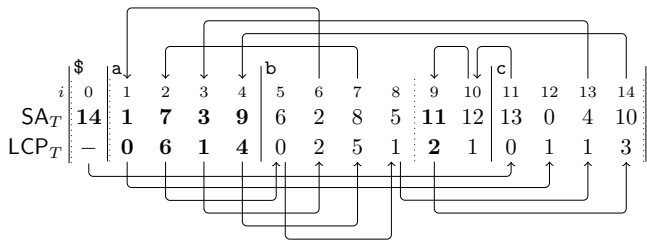


Figure 1: Example of the inducing steps on the string $T =$ [cabacbbabacbbc$]. Assume that the relative order of S*-suffixes (bold font) is known (see Figure 3 for the recursion). They are placed into their corresponding S-subbuckets as described in step (2). In step (3), suffix- and LCP-values of the L-suffixes (normal font) are induced from the LCPs of S*-suffixes. Afterwards, in step (4), the reverse process (shown above the array) induces all S-suffixes from the L*-suffixes.

are those suffixes T_i with $T_{i+1} \in \mathcal{L}$. So T_i can be inserted into $\mathcal{L}$ as the next larger L-suffix among all suffixes that start with t_i. Thus the iterative procedure always picks the smallest remaining suffix and places it as the next larger one in the t_i-L-subbucket. 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. This results in the following high-level four step algorithm SAIS [Nong et al., 2011]:

(1) Sort the S*-suffixes. This step will be explained in more detail below.

(2) Put the sorted S*-suffixes into their corresponding S-subbuckets, without changing their order. All other entries remain undefined. Prepare head and tail pointers for all L-subbuckets in SA.

(3) Induce the order of the L-suffixes by scanning SA from *left to right* (skipping undefined entries): for every position i in SA, if $T_{\mathsf{SA}[i]-1}$ is L-type, write $\mathsf{SA}[i]-1$ to the current head of the c-L-subbucket (where $c = t_{\mathsf{SA}[i]-1}$, the preceding character), and increase the current head of that bucket by one. Note that this step can only induce "to the right" (the current head of the c-L-subbucket is larger than i).

(4) Induce the order of the S-suffixes by scanning SA from *right to left*: for every position i in SA, if $T_{\mathsf{SA}[i]-1}$ is S-type, write $\mathsf{SA}[i]-1$ to the current *tail* of the c-S-subbucket ($c = t_{\mathsf{SA}[i]-1}$), and *de*crease the current tail of that bucket by one. Note that this step can only induce "to the left," and might intermingle arbitrary S-suffixes with S*-suffixes.

Figure 1 illustrates the inducing steps (3) and (4) with arrows, step (3) as arrows below the suffix array and step (4) above it (ignore for now the row labeled "LCP_T").

It remains to find the relative order of S*-suffixes. For each S*-suffix T_i, we define the S*-*substring* $[t_i, \ldots, t_j]$, 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 letter 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. 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 of lexicographic names of S*-substrings [Nong et al., 2011]. The overlapping character is

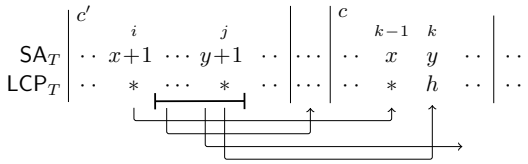


Figure 2: General scheme of the inducing step. When inducing k, the LCP value $h = \text{RMQ}_{\mathsf{LCP}_T}(i+1, j) + 1$ can be derived using an range minimum query (RMQ) between the previous and current relative ranks of the induce sources.

needed because all L- and then all S-suffixes encompassed by the S*-substring are induced from the recursively determined order, thus the last S-suffix (the overlapping character) must also be considered. Throughout this article, we denote the reduced string consisting of lexicographic names by R, and the recursively computed suffix array by SA_R.

3 Inducing LCP-Arrays in Main Memory

In this section we explain how the induced sorting algorithm (Section 2.1) can be modified to also compute the LCP-array *in main memory*. This will form the basis of our *external memory* adaption in Section 6. Besides that, the algorithm is novel and an interesting alternative to existing internal memory LCP-array construction algorithms [Kasai et al., 2001; Manzini, 2004; Kärkkäinen et al., 2009; Gog and Ohlebusch, 2011].

The basic idea is that whenever we place two S- or L-suffixes T_x and T_y at adjacent places $k-1$ and k in same c-bucket of the final suffix array (see Figure 2 and steps (3)–(4) of the algorithm in Section 2.1), the length of their longest common prefix can be induced from the longest common prefix of the suffixes T_{x+1} and T_{y+1}. As the latter suffixes are exactly those that caused the inducing of T_x and T_y, we already know their LCP-value ℓ (by the order in which we fill SA), and can hence set $\mathsf{LCP}_T[k]$ to $\ell + 1$.

The details are described next. We augment the steps of the induced sorting algorithm as follows:

(1′) Compute LCP_{S^*}, the array of LCP-values of the S*-suffixes (see Section 3.1).

(2′) Whenever we place an S*-suffix into its S-subbucket, we also store its LCP-value (as computed in step (1′)) at the corresponding position in LCP_T.

(3′) Suppose that the j-th inducing iteration just put suffix T_y with $y = \mathsf{SA}[k]$ into its c-L-subbucket ($c = t_y$) at position k. If T_y is the first suffix in its L-subbucket, we set $\mathsf{LCP}_T[k]$ to 0. Otherwise, suppose further that in a previous iteration $i < j$ the inducing step placed suffix T_x at $k-1$ in the same c-L-subbucket, with $x = \mathsf{SA}[k-1]$. Then if i and j are in different buckets, the corresponding suffixes T_{x+1} and T_{y+1} start with different characters, so we set $\mathsf{LCP}_T[k]$ to 1, as the suffixes T_x and T_y share only the common character c at their beginnings. Otherwise, $x+1$ and $y+1$ are in the same c'-bucket, with $t_{x+1} = c' = t_{y+1}$. Then the length h of the longest common prefix of the suffixes T_{x+1} and T_{y+1} is given by the *minimum* value in $\mathsf{LCP}_T[i+1, j]$, all of which are in the same c'-bucket and have therefore already been computed in previous iterations. We can hence set $\mathsf{LCP}_T[k]$ to $h + 1$. We address the problem of how to compute these minima in Section 3.2.

(4′) This step is symmetric to step (3′).

We will resolve the problem of computing the LCP-value between the last L-suffix and the first S-suffix in a bucket in Section 3.3.

For an example, look at the inducing of suffixes T_2 and T_8 in Figure 1. Both suffixes start with character b. The suffixes that caused their inducing are T_3 and T_9 at positions 3 and 4 of SA_T, respectively, both starting with a. Their LCP is 4, which is (trivially) determined by finding the minimum in $\mathsf{LCP}_T[4,4]$. Therefore, we set $\mathsf{LCP}_T[7]$ to 5.

3.1 Computing LCP-Values of S*-suffixes

Here, we give the details on step (1′) above. 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.

Let $s^*_1, \ldots, s^*_K$ be the K positions of S*-substrings in T, ordered as in the input string. Given the recursively calculated LCP array LCP_R and SA_R, we now 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 S*-suffixes $s^*_{\mathsf{SA}_R[k-1]}$ and $s^*_{\mathsf{SA}_R[k]}$. See also Figure 3, which gives an example of all concepts presented in this section.

There are two main issues to be dealt 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 3 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, 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, with K being the number of S*-substrings and $s^*_K = n-1$. Also, during the lexicographic naming process (which sorts the S*-substrings), we compute the LCPs of lexicographically consecutive S*-substrings in an array LCP_N. The resulting array LCP_N is then prepared for constant-time range minimum queries (RMQs) [Fischer and Heun, 2011]; such queries return the minimum among all array entries in a given range: $\text{RMQ}_A(\ell, r) = \min_{\ell \le i \le r} A[i]$ for an array A. This allows us to find the common characters of arbitrary S*-suffixes, as shown in the next lemma.

Lemma 2. *Given* SA_R, ISA_R, LCP_R, Size_{S^*}, *and* LCP_N, *the array* $\mathsf{LCP}_{S^*}[k]$ *can be calculated by*

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

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

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

Proof. We must show that this expression counts the maximum number of equal characters starting at the S*-suffixes $s^*_{\mathsf{SA}_R[k-1]}$ and $s^*_{\mathsf{SA}_R[k]}$, which are lexicographically consecutive. Because they are consecutive, $\mathsf{LCP}_R[k]$ was calculated recursively as the number of equal *complete* S*-substrings starting at these positions. Thus summing over the sizes of those equal S*-substring entries from $\mathsf{SA}_R[k]$ to $\mathsf{SA}_R[k] + \mathsf{LCP}_R[k] - 1$ (or, equivalently, $\mathsf{SA}_R[k-1]$ to $\mathsf{SA}_R[k-1] + \mathsf{LCP}_R[k] - 1$) yields the total number of equal characters in whole S*-substrings.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	b	a	c	b	b	a	b	a	c	b	b	c	$
type(i)	L	S*	L*	S*	L*	L	L	S*	L*	S*	L*	S*	S	L*	S*
R		1			2			1		3			4		0
Size_{S^*}		2			4			2		2			3		0

k	0	1	2	3	4	5
SA_R	5	0	2	1	3	4
LCP_R	–	0	1	0	0	0
LCP_{S^*}	–	0	6	1	4	0

$[t_i \dots t_j]$	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 3: 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^*}, consisting of the sorted S*-substrings and associated information. The last column LCP_N shows the LCPs of lexicographically consecutive S*-substrings.

It thus remains to determine the longest common prefix of the first pair of unequal S*-substrings contained in both S*-suffixes. This first pair of unequal S*-substrings is $\mathsf{SA}_R[k-1]+\mathsf{LCP}_R[k]$ and $\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]$. However, instead of calculating $\text{LCP}_T(\mathsf{SA}_R[k-1]+\mathsf{LCP}_R[k], \mathsf{SA}_R[k]+\mathsf{LCP}_R[k])$ directly, we resort to looking up the lexicographic ranks of these positions in ISA. So, $\mathsf{ISA}[\mathsf{SA}_R[k-1]+\mathsf{LCP}_R[k]]$ and $\mathsf{ISA}[\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]]$ are the lexicographic ranks of the pair of unequal S*-substrings. These ranks need not be adjacent in SA, therefore instead of a direct lookup in LCP_N, an RMQ between these ranks becomes necessary. Notice that LCP_N is constructed from names, while the queries boundaries are ranks. This is however still correct, as the range in LCP_N corresponding to the same lexicographic name is filled with the length of the name, except for the first entry. Because the LCP to both predecessor or successor name is no longer than the length, for RMQ calculation is suffices to take any rank of the same lexicographic name.

If $\mathsf{LCP}_R[k]=0$, then the whole expression reduces to $\mathsf{LCP}_N[k]$, as one would expect. □

We point out a fine detail about LCP_N here: in SAIS, letters of S*-substrings are compared first by character and then by type. For LCP construction, however, we must count equal characters even though they may have *different* types. Consider two such suffixes that both start with equal characters but different types. We can calculate their LCP by considering only the number of *repetitions* of the equal character. 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 obviously that character must be different. Thus the LCP of the two considered suffixes is the minimum of the two repetition counts. For occurrences within an S*-substring the repeating letters can be counted directly. But, for cases where the equal sequence extends beyond the end of an S*-substring, we save the repetition count of the overlapping character explicitly to handle this case.

For example, regard the penultimate row on the right side of Figure 3. Even though there are only 3 common characters in [acb] and its preceding S*-substring [acbba], for the calculation in Lemma 2 to be correct, there must be a '4' in LCP_N. This LCP-value can be deduced from the repetition count '1' of the shorter string [acb], which matches the second 'b' of the longer string [acbba].

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.

3.2 Finding Minima

To find the minimum value in $\mathsf{LCP}[i+1, j]$ or $\mathsf{LCP}[j+1, i]$ (steps $(3')$ and $(4')$ above), we have several alternatives. Let us focus on the left-to-right scan (step $(3')$); the right-to-left scan (step $(4')$) is symmetric. The simplest idea is to scan the whole interval in LCP; this results in overall $O(n^2)$ running time. A better alternative would be to keep an array M of size $|\Sigma|$, such that the minimum is always given by $M[c]$ if we induce an LCP-value in bucket c. More formally, we define the array $M[1, |\Sigma|]$ by $M[c'] := \min \mathsf{LCP}[i_{c'} + 1, j]$, where $c' \in \Sigma$ and $i_{c'} \leq j$ is the last position from where we induced to the c'-bucket. To keep M up-to-date, before retrieving $h = \text{RMQ}_{\mathsf{LCP}_T}(i + 1, j) + 1$ from $M[c]$, we update all entries in M that are larger than $\mathsf{LCP}[j]$ by $\mathsf{LCP}[j]$, since their corresponding range minimum queries overlap with position j. Finally, we set $M[c] = +\infty$; this ensures that in the next iteration $j + 1$ the value $M[c]$ will be set correctly. In total, this approach has $O(n|\Sigma|)$ running time. A further refinement of this technique stores the values in M in sorted order and uses binary search on M to find the minima, similar to the stack used by Gog and Ohlebusch [2011]. This results in overall $O(n \lg |\Sigma|)$ running time.

Yet, we can also update the minima in $O(1)$ amortized running time, as explained next. Recall that the queries lie within a single bucket (called c'), and every bucket is subdivided into an L- and an S-subbucket. The idea is to also subdivide the query into an L- and an S-query, and return the minimum of the two. The S-queries are simple to handle: in step $(3')$, only $\mathtt{S}^*$-suffixes will be scanned, and these are static. Hence, we can preprocess every $\mathtt{S}^*$-subbucket (consisting of $\mathtt{S}^*$-suffixes starting with the same character) with a static data structure for constant-time range minima, using overall linear space [Fischer, 2010, Thm. 1]. The L-queries are more difficult, as elements keep being written to them during the scan. However, these updates occur in a very regular fashion, namely in a left-to-right manner. This makes the problem simpler: we maintain a *LRM-tree* [Barbay et al., 2012, Def. 1] $\mathcal{M}_{c'}$ for each bucket c', which is initially empty (no L-suffixes written so far). When a new L-suffix along with LCP-value $\ell + 1$ is written into its c-bucket, we climb up the rightmost path of $\mathcal{M}_c$ until we find an element x whose corresponding array-entry is strictly smaller than $\ell + 1$ ($\mathcal{M}_c$ has an artificial root holding LCP-value $-\infty$, which guarantees that such an element always exists). The new element is then added as x's new rightmost leaf; an easy amortized argument shows that this results in overall linear time. Further, $\mathcal{M}_c$ is stored along with a data structure for constant-time *lowest common ancestor queries* (LCAs) which supports dynamic leaf additions in $O(1)$ worst-case time [Cole and Hariharan, 2005]. Then the minimum in any range in the processed portion of the L-subbucket can be found in $O(1)$ time [Fischer, 2010, Lemma 2].

What we have described in the preceding paragraph was actually more general than what we really needed: a solution to the *semi-dynamic range minimum query problem* with constant $O(1)$ query- and amortized $O(1)$ insertion-time, with the restriction that new elements can only be appended at the end (or beginning, respectively) of the array. Our solution might also have interesting applications in other problems. In our setting, though, the problem is slightly more specific: the sizes of the arrays to be prepared for RMQs are known in advance (namely the sizes of the L- or S-subbuckets); hence, we can use any of the (more practical) preprocessing-schemes for (static) RMQs in $O(1)$ worst-case time [Fischer and Heun, 2007; Alstrup et al., 2004], and update the respective structures, which

are essentially precomputed RMQs over suitably-sized blocks, whenever enough elements have arrived.

3.3 Computing LCP-values at the L/S-Seam

There is one subtlety in the above inducing algorithm we have withheld so far, namely that of computing the LCP-values between the last L-suffix and the first S-suffix in a given c-bucket (we call this position the L/S-*seam*). More precisely, when reaching an L/S-seam in step (3′), we have to re-compute the LCP-value between the first S*-suffix in the c-bucket (if it exists) and the last L-suffix in the same c-bucket (the one that we just induced), in order to induce correct LCP-values when stepping through the S*-suffixes in subsequent iterations. Likewise, when placing the very first S-suffix in its c-bucket in step (4′), we need to compute the LCP-value between this induced S-suffix and the largest L-suffix in the same c-bucket. (Note that step (4) might place an S-suffix before all S*-suffixes, so we cannot necessarily re-use the LCP-value computed at the L/S-seam in step (3′).)

The following lemma shows that the LCP-computation at L/S-seams is particularly easy:

Lemma 3. *Let T_i be an* L*-suffix, T_j an* S*-suffix, and $t_i = c = t_j$ (the suffixes are in the same c-bucket in* SA*). Further, let $\ell \geq 1$ denote the length of the longest common prefix of T_i and T_j. Then*

$$[t_i \dots t_{i+\ell-1}] = c^\ell = [t_j \dots t_{j+\ell-1}] \ .$$

Proof. Assume that $t_{i+k} = c' = t_{j+k}$ for some $2 \leq k < \ell$ and $c' \neq c$. Then if $c' < c$, both T_i and T_j are of type L, and otherwise ($c' > c$), they are both of type S. In any case, this is a contradiction to the assumption that T_i is of type L, and T_j of type S. □

In words, the above lemma states that the longest common prefix at the L/S-seam can only consist of equal characters. Therefore, a *naive* computation of the LCP-values at the L/S-seam is sufficient to achieve overall linear running time in main memory: every character t_i contributes at most to the computation at the L/S-seam in the t_i-bucket, and not in any other c-bucket for $c \neq t_i$.

4 Induced Suffix Sorting in External Memory

We now design an EM algorithm based on the induced sorting principle that runs in sorting complexity and has a lower constant factor than DC3 [Dementiev et al., 2008a]. The basis for this algorithm is an efficient EM priority-queue (PQ) [Dementiev et al., 2008b], as suggested by the proof of Lemma 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. Section 4.1 is then devoted to complications that arise due to large S*-substrings.

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 (corresponding to step (1) of the high-level algorithm in Section 2.1). 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 (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 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

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) ]
```

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 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 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.

4.1 Splitting Large Tuples

After the detailed description of Algorithm 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 Algorithm 1, the tuples would grow larger than an I/O block B, and one would have to resort to long string sorting [Arge et al., 1997]. More importantly, 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, once splitting S*-substrings from their beginning and the second time from their end.

Long string sorting in EM can be dealt with using lexicographic naming and doubling [Arge et al., 1997, Section 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 Algorithm 1. This is done by dividing the S*-substrings into *split substrings* of length at most B, starting at their *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 the overlapping character need not be S*-type. Thus split tuples are distinct from ordinary S*-substrings and the recursive super-alphabet $\Sigma' = (\Sigma \times \{\mathtt{L}, \mathtt{S}\})^*$ (each character of the reduced string corresponds to a split substring, within which each character has a letter and a type). After the recursive call, long S*-substrings are correctly ordered among all other S*-substrings due to suffix sorting, and split tuples can easily be discarded in line 10 as they do not correspond to any S*-suffix. The *d-critical* version of SAIS [Nong et al., 2011, Section 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. Our solution is to place a limit on the number of characters stored in the PQ, and fetch additional characters when needed. Since the characters in the PQ tuples are ordered in reverse, we must again split S*-substrings, but this time from their *end*. We call the items containing the last D_0 characters of an S*-substring the *seed tuples*, and all items containing additional (up to D) characters *continuation tuples*. When the currently processed PQ tuple requires additional characters, we say it *underruns*.

The challenge in EM is to have the additional characters readily available when needed, since we cannot spend an I/O to fetch each continuation tuple. We solve this by noting that we can predict *when* a continuation tuple is required. The additional characters are needed exactly at the *boundaries between repetition buckets* (see Section 2.1 for the definition of repetition buckets). To understand this, consider what happens when a tuple underruns. The point is that we need not fetch the missing characters immediately, since the earliest output position which may change due to the additional characters lies in the next *repetition* bucket. This occurs when the characters in the continuation tuple themselves induce into the current bucket. Thus we can postpone matching of continuation tuples to underrun tuples to the boundaries between repetition buckets. We have thus established time points when underrun tuples must be matched, however, this also implies *which* tuples are matched at these boundaries. We can thus pre-sort the set of continuation tuples by repetition bucket

ALGORITHM 2: Inducing step with $\mathtt{S}^*$-substrings split by D_0 and D, replacing lines 10–15 of Algorithm 1

1 $\mathcal{D} := \{\, s^*_k - D_0 - \nu \cdot D \mid \nu \in \mathbb{N},\, s^*_k - D_0 - \nu \cdot D > s^*_{k-1},\, k \in [0, K) \,\}$ *// split positions, with* $s^*_{-1} = 0$
2 $S^* := \mathsf{Sort}[\,(t_j, \mathsf{ISA}_R[k], [t_{j-1} \dots t_i], j, \mathbb{1}_{i \in \mathcal{D}}) \mid j = s^*_k,\, i = \max(s^*_{k-1}, j - D_0),\, k \in [0, K)\,]$
3 $L := \mathsf{Sort}[\,(t_j, \mathrm{rep}(j), j, [t_{j-1} \dots t_i], \mathbb{1}_{i \in \mathcal{D}}) \mid j \in \mathcal{D},\, i = \max(s^*_{k-1}, j - D),\, t_j \text{ is } \mathtt{L}\text{-type}\,]$
4 $S := \mathsf{Sort}[\,(t_j, \mathrm{rep}(j), j, [t_{j-1} \dots t_i], \mathbb{1}_{i \in \mathcal{D}}) \mid j \in \mathcal{D},\, i = \max(s^*_{k-1}, j - D),\, t_j \text{ is } \mathtt{S}\text{-type}\,]$
5 $\rho_L := 0, \quad a := \bot, \quad r_a = 0, \quad S^* := \mathsf{Stack}(S^*), \quad Q_L := \mathsf{CreatePQ}(\emptyset \text{ by } (t_i, r, [t_{i-1} \dots t_{i-\ell}], i, c))$
6 **while** Q_L.NotEmpty() or S^*.NotEmpty() **do**
7 **while** Q_L.Empty() or $t < Q_L$.TopChar() with $(t, r, [t_{i-1} \dots t_{i-\ell}], i, c) = S^*$.Top() **do**
8 Q_L.insert$(t_{i-1}, \rho_L{+}{+}, [t_{i-2} \dots t_{i-\ell}], i-1, c)$, S^*.Pop() *// induce from* $\mathtt{S}^*$*-suffixes*
9 $a' := a,\ a := Q_L$.TopChar(), $r_a := (r_a + 1)\mathbb{1}_{a'=a},\ m := \rho_L,\ M := \emptyset$ *// next a-repetition bucket*
10 **while** Q_L.TopChar() $= a$ and Q_L.TopRank() $< m$ **do** *// induce from* L*-suffixes*
11 $(t_i, r, [t_{i-1} \dots t_{i-\ell}], i, c) = Q_L$.extractMin(), A_L.append$((t_i, i))$ *// save* i *as next* L*-type*
12 **if** $\ell > 0$ **then**
13 **if** $t_{i-1} \geq t_i$ **then** Q_L.insert$(t_{i-1}, \rho_L{+}{+}, [t_{i-2} \dots t_{i-\ell}], i-1, c)$ *//* T_{i-1} *is* L*-type*
14 **else** L^*.append$(\,(t_i, \rho_L{+}{+}, [t_{i-1} \dots t_{i-\ell}], i, c)\,)$ *//* T_{i-1} *is* S*-type*
15 **else if** $\ell = 0$ and $c = 1$ **then** M.append$(i, \rho_L{+}{+},)$ *// need continuation?*
16 **foreach** Merge$([\,(a, r_a, i, r) \mid (i, r) \in \mathsf{Sort}(M)\,]$ with $(a, r_a, i, [t_{i-1}, \dots, t_{i-\ell}], c) \in L)$ **do**
17 **if** $t_{i-1} \geq t_i$ **then** Q_L.insert$(t_{i-1}, r, [t_{i-2} \dots t_{i-\ell}], i-1, c)$ *//* T_{i-1} *is* L*-type*
18 **else** L^*.append$(\,(a, r, [t_{i-1} \dots t_{i-\ell}], i, c)\,)$ *//* T_{i-1} *is* S*-type*

(and text position) and have them readily available for merging with underrun tuples.

This procedure is the key idea of Algorithm 2, which replaces lines 10–15 of Algorithm 1 and which we 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 $\mathtt{S}^*$-suffix until the preceding $\mathtt{S}^*$-suffix is met ($D_0 \geq D$ indicates when to split at all, and $D \geq 1$ being the split length of continuation tuples). As before, for each $\mathtt{S}^*$-substring a seed tuple is stored in the S^* array, except that only the initial D_0 payload characters are copied. If an $\mathtt{S}^*$-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 at repetition bucket boundaries; $\mathrm{rep}(i)$ 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.

With these different sources of characters pre-computed, we have to break up the elegant while loop of Algorithm 1 into three separate phases: (1) inducing from $\mathtt{S}^*$-suffixes in lines 7–8, (2) inducing from L-suffixes in lines 10–15, and (3) finding continuation tuples for underrun PQ items in lines 16–18. Since we must match continuation tuples at each repetition bucket boundary, one iteration of the large while loop (lines 6–18) is designed to induce all items of one repetition bucket. An additional difference from Algorithm 1 is that in line 5 the PQ is initialized as empty and S^* will be processed as a stack.

More details of Algorithm 2 are described next. The two induction sources, the S^* and L arrays, are alternated between, with precedence depending on their top character: Q_L.TopChar() $:= t_i$ with $(t_i, r, \tau, i, c) = Q_L$.Top(). Since L-suffixes are smaller than $\mathtt{S}^*$-suffixes if they start with the same character, the while loop in 7–8 may only induce from $\mathtt{S}^*$-suffixes with the first character being smaller than Q_L.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

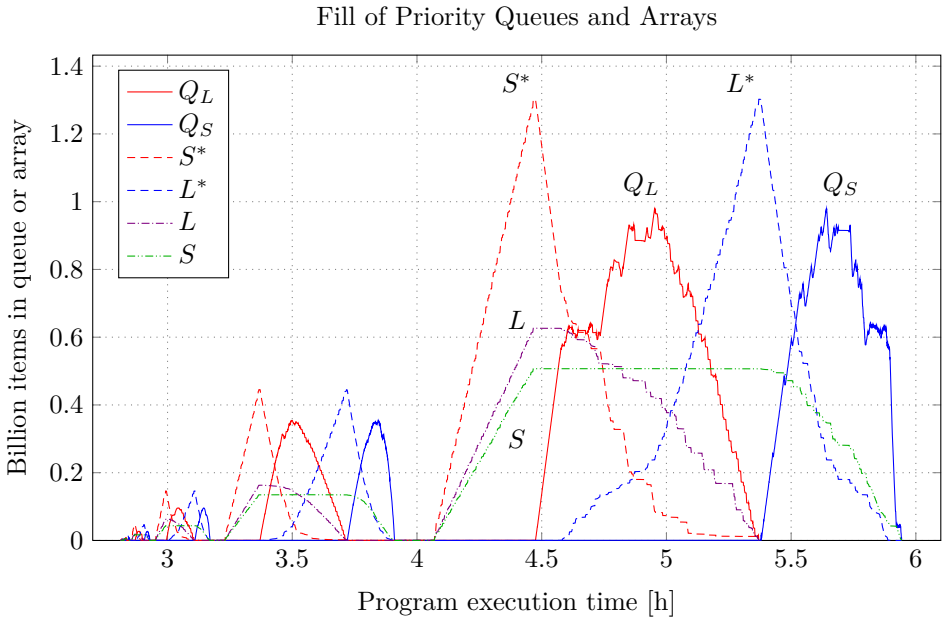


Figure 4: The graph shows the number of items in the six main data structures used in Algorithm 1, plotted over the program execution time of one run of our eSAIS implementation on 4 GiB of Wikipedia input.

11–14 the extracted tuple is handled as in Algorithm 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 Section 5 we compute the optimal values for D_0 and D, and analyze the resulting I/O volume.

4.2 Fill of Priority Queues and Arrays in an Example Program Run

In this section we give a visual insight into the eSAIS algorithm using the example of the plot in Figure 4. The graph shows the number of items contained in the two PQs and the most important four arrays for an example run of our eSAIS implementation on 4 GiB of Wikipedia XML (see Section 7 for details on the implementation, input, and experimental setup).

One can see the unwinding of four recursive levels, each composed of the inducing process described in Algorithm 1, lines 10–16, and augmented by Algorithm 2. In the first phase (lines 1–4 of Algorithm 2), the arrays S^*, L and S are simultaneously constructed by reading the input and the recursively calculated ISA_R. Thereafter, the while-loop in lines 6–18 runs until both Q_L and S^* are empty. In this phase, all L-suffixes are ordered. The array L

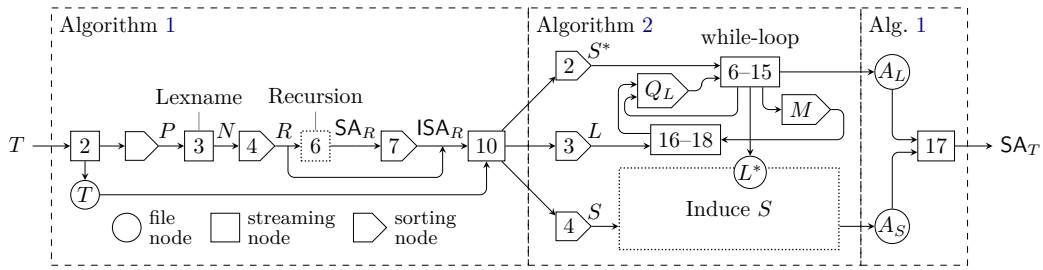


Figure 5: Data flow graph of the algorithm; numbers refer to the line numbers of Algorithm 1 and Algorithm 2, respectively. The input T is read and saved to a file (2), while creating tuples. Sorting these tuples yields P, whose entries are lexicographically named in N (3) and sorted again by string 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 Algorithm 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 L-suffix order information into A_L, stores merge requests to M when tuples underrun, reinserts a shortened tuple in Q_L, or 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 L-suffixes finishes, the process is repeated with seed tuples from $\mathtt{L}^*$ and continuation tuples from S, yielding the final S-suffix order values in A_S. The output suffix array is constructed by merging A_L and A_S (17).

contains continuation tuples for tuples that underrun when processing Q_L, thus L is fully consumed when Q_L is empty. While processing L-suffixes, the while-loop outputs the array L^* in line 14. These tuples are the seeds for the symmetric while-loop, which orders all S-suffixes using Q_S and consuming S.

Notice that the peak fill of arrays S^* and L^* is the same. This corresponds to the number of $\mathtt{S}^*$-substrings, as each substring contains exactly one $\mathtt{S}^*$- and one $\mathtt{L}^*$-character, except for the first and last. The irregular fill of Q_L and Q_S is due to the particular real-world input. It shows an uneven distribution of the ASCII characters in the English text: the short plateaus in Q_L and Q_S are probably due to the large number of spaces.

5 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 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. 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.

For our practical experiments we assume $M < n \leq \frac{M^2}{B}$, 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. This assumption will be used in the following analysis only once, when a relation between Sort and Scan is required to calculate a practical value for D and D_0.

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. The overlapping character is counted separately. For further simplicity, we assume that line 15 of Algorithm 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 Algorithm 1 (including Algorithm 2) into a pipelined data flow graph in Figure 5.

Lemma 4. *To minimize I/O cost Algorithm 2 should use $D = 3$ and $D_0 = 8$ for splitting $\mathtt{S}^*$-strings, when $n \leq \frac{M^2}{B}$.*

Proof. We first focus on the number of elements sorted and scanned by the algorithm for one long $\mathtt{S}^*$-substring of 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 1. *For a string of length n the I/O volume of Algorithm 1 is bounded by $\textsc{Sort}(17n)$ + $\textsc{Scan}(9n)$, when splitting with $D=3$ and $D_0=8$ in Algorithm 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 $2\leq\ell\leq n$, where $\ell=2$ is the smallest possible length of $\mathtt{S}^*$-substrings, due to exclusion of the overlapping character. Algorithm 1 needs $\textsc{Scan}(2n)$ to read T twice (in lines 2 and 10) and $\textsc{Sort}(n+\frac{n}{\ell}\cdot 2)$ to construct P in line 2, counting the overlapping character and excluding the boolean type, which can be encoded into i. In this $\textsc{Sort}$ the I/O volume of $\mathtt{Lexname}_{S^*}$ is already accounted for. Creating the reduced string R requires sorting of N, and thus $\textsc{Sort}(2\cdot\frac{n}{\ell})$ I/Os. Then the suffix array of the reduced string R with $|R|\leq\frac{n}{\ell}$ is computed recursively and inverted using $\textsc{Sort}(2\cdot\frac{n}{\ell})$, or the names are already unique. After creating ISA_R, Algorithm 2 is used with the parameters derived in Lemma 4, 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 $\textsc{Scan}(2n)$. In sum this is

$$\begin{aligned} V(n) \leq\ & \textsc{Scan}(2n)+\textsc{Sort}(n+\tfrac{n}{\ell}\cdot 2)+\textsc{Sort}(\tfrac{n}{\ell}\cdot 2)+V(\tfrac{n}{\ell}) \\ & +\textsc{Sort}(\tfrac{n}{\ell}\cdot 2)+\textsc{Scan}(2n)+\tfrac{n}{\ell}\cdot\min\{\textsc{Sort}(\tfrac{23}{3}\ell-3)+\textsc{Scan}(5)\,, \\ & \qquad\qquad \textsc{Sort}(\tfrac{1}{2}\ell(\ell+1)+\ell\cdot 3)+\textsc{Scan}(\tfrac{\ell}{2})\}\,. \end{aligned}$$

ALGORITHM 3: EM calculation of the longest common prefix array of $\mathtt{S}^*$-suffixes.

1 $\mathsf{SALCP}_R := \mathtt{eSAIS\text{-}LCP}(R)$ and ISA_R calculated from SA_R
2 $Q_1 := \mathtt{Sort}[\,(\mathsf{SA}_R[k]-1,k),(\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]-1,k)\mid$ *// range sum over* Size_{S^*}
3 $\quad(\mathsf{SA}_R[k],\mathsf{LCP}_R[k])\in\mathsf{SALCP}_R$ with $\mathsf{LCP}_R[k]>0\,]$
4 $A_1 := \mathtt{Sort}[\,(k,\sum_{i=0}^{s}\mathsf{Size}_{S^*}[i])\mid \mathtt{Merge}((s,k)\in Q_1$ and $(s,\sum_{i=0}^{s}\mathsf{Size}_{S^*}[i])\in\mathtt{PrefixSum}(\mathsf{Size}_{S^*}))\,]$
5 $Q_2 := \mathtt{Sort}[\,(\mathsf{SA}_R[k-1]+\mathsf{LCP}_R[k],k),(\mathsf{SA}_R[k]+\mathsf{LCP}_R[k],k)\mid$ *// batched random access on* ISA_R
6 $\quad((\mathsf{SA}_R[k-1],\mathsf{LCP}_R[k-1]),(\mathsf{SA}_R[k],\mathsf{LCP}_R[k]))=(\mathsf{SALCP}_R[k-1],\mathsf{SALCP}_R[k])\,]$
7 $A_2 := \mathtt{Sort}[\,(k,\mathsf{ISA}_R[p])\mid \mathtt{Merge}((p,k)\in Q_2$ and $(p,\mathsf{ISA}_R[p])\in\mathsf{ISA}_R)\,]$
8 $Q_3 := [\,\mathrm{RMQ}(\ell+1,r,k)\mid((k,\ell),(k,r))=(A_2[i],A_2[i+1])\,]$ *// RMQs on* LCP_N
9 $A_3 := [(k,\mathrm{RMQ}_{\mathsf{LCP}_N}(\ell,r)]=\mathtt{AnswerRMQ}(Q_3,\mathsf{LCP}_N)$
10 $\mathsf{LCP}_{S^*} := [\,(s_2-s_1)+m\mid\mathtt{Merge}((k,s_1)=A_1[i],\,(k,s_2)=A_1[i+1]$ and $(k,m)=A_3[j])\,]$

Maximizing $V(n,\ell)$ for $2\le\ell\le n$ by $\ell=2$, we get $V(n,\ell)\le V(n,2)\le \textsc{Sort}(8.5n)+\textsc{Scan}(4.5n)+V(\frac{n}{2})$ and, solving the recurrence, $V(n,\ell)\le\textsc{Sort}(17n)+\textsc{Scan}(9n)$. In Section 7 a worst-case string is constructed with $\mathtt{S}^*$-substrings of length $\ell=2$ on every recursive level. □

The proof of Theorem 1 does not need to assume $n\le\frac{M^2}{B}$, since we take D and D_0 as fixed parameters independent of n. These D and D_0 give minimal I/O cost under our practical assumptions, yet Theorem 1 holds whether of not these parameters are optimal.

6 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 in Section 3; here, we adapt it to work in external memory.

6.1 Calculating LCP_{S^*}

The pseudo-code is shown in Algorithm 3, where Q_1,Q_2,Q_3 are sets of queries, and A_1,A_2,A_3 their respective answers. Line 1 recursively calculates SA_R and LCP_R. According to Lemma 2, two subproblems must be solved efficiently in external memory: range sums over Size_{S^*} (lines 2–4), and range minimum queries over LCP_N (line 5–9). 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 LCP_N, which delivers $\sum_{k=0}^{\mathsf{SA}_R[k]-1}\mathsf{LCP}_N$ and $\sum_{k=0}^{\mathsf{SA}_R[k]+\mathsf{LCP}_R[k]-1}\mathsf{LCP}_N$, from which the range sum is easily calculated.

For the static range minimum queries in LCP_N, we follow a common RAM-technique [Fischer and Heun, 2011]: 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 [Kärkkäinen et al., 2006].

6.2 Computing LCPs by Finding Minima

The RMQs from Section 3.2 delivering the LCP values are created in batch while inducing SA and answered afterwards, forming the LCP array. This is possible, as the indexes i and j in $\mathrm{RMQ}_{\mathsf{LCP}_T}(i+1,j)+1$ are the relative ranks 'ρ_L' of two consecutively extracted tuples from

the PQ Q_L (and symmetrically for the second phase). Notice that the first while-loop in Algorithm 1 orders only the L-suffixes in SA. Likewise, the batch process computes only all LCP-values of L-type suffixes. The corresponding RMQs are calculated on a virtual array, denoted by $\mathsf{LCP}_T|_{Q_L}$, which interleaves the entries of LCP_{S^*} with LCP-values of L-suffixes bucket-wise, and is indexed by the relative rank ρ_L.

As we saw in Section 3.2, solving the RMQs on $\mathsf{LCP}_T|_{Q_L}$ is in fact a semi-dynamic problem. To solve it, we decided not to explore which of the well known EM data structures such as buffer trees [Arge, 2003] 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 Section 5.

Under this assumption we can split the array $\mathsf{LCP}_T|_{Q_L}$ into *blocks* of size $s := C \cdot M$ and keep the $\mathsf{LCP}_T|_{Q_L}$-values of the current block in main memory. 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 Section 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 $\mathsf{LCP}_T|_{Q_L}$-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, etc. Their LCP-values are 0,1,2, etc., which is exactly their repetition count. The current repetition count, however, is the readily-available 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 (see Section 3.2).

As already discussed in Section 3.3, the LCP-value at the L/S-seam requires special consideration. For handling the seam in EM we reapply Lemma 3 in a different manner: for each c-bucket we save the maximum repetition count in the L-subbucket during the first while-loop. Then, when inducing S-suffixes in the symmetric while-loop, the L/S-seam LCP-value can be determined from the maximum repetition count in L- and S-subbucket. As suggested by Lemma 3, the true value is the smaller of both repetition counts.

7 Experimental Evaluation

We implemented the eSAIS algorithm with integrated LCP construction in C++ using the external memory library STXXL [Dementiev et al., 2008b]. This library provides efficient external memory sorting and a priority queue that is modeled after the design for cached memory [Sanders, 2000]. 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 does not directly apply. We found that $D = D_0 = 3$ are good splitting values in practice, which match the theoretical average $\mathtt{S}^*$-substring length. All results of the algorithms were verified using a suffix array checker [Dementiev et al., 2008a, Section 8] and a semi-external version of Kasai's LCP algorithm [Kasai et al., 2001] (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 [Kärkkäinen and Rantala, 2009] is used. Strings of larger data types are sorted in RAM using `gcc-4.4` 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 was obtained from http://dumps.wikimedia.org/; our dump is dated `enwiki-20120601`.

Gutenberg is a concatenation of all ASCII text document files 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 uppercase 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 `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 $\Sigma = [\,\mathtt{\$}, \sigma_1, \ldots, \sigma_p\,]$ and the grammar $\{S \to T_1\mathtt{\$},\ T_i \to T_{i+1}\sigma_i T_{i+1}$ for $i = 1, \ldots, p-1$ and $T_p \to \sigma_p\}$. For $p = 4$ and $\Sigma = [\,\mathtt{\$}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\,]$, we get `dcdbdcdadcdbdcd$`; for the test runs we replaced `$` with σ_0. The name "Skyline" comes from the corresponding height diagram, which looks like a metropolitan skyline when drawn with smallest character on top.

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. All four are "Seagate Barracuda 7200.10 ST3250820AS" disks. A single disk's write and read throughput ranges between 80 MiB/s on the outside and 72 MiB/s on the inside. Parallel I/O speed to the four disks ranges between 320 MiB/s and 240 MiB/s, and was measured using STXXL's `benchmark_disks` tool. 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. The disks were labeled "Seagate SV35.5 ST31000525SV" and an individual disk's throughput ranged from 110 MiB/s to 90 MiB/s. All disks together reached at most 520 MiB/s and on average 450.0 MiB/s when writing 4 TiB of data.

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

7.1 Plain Suffix Array Construction

As noted in the introduction, the previously fastest EM suffix sorter is DC3 [Dementiev et al., 2008a]. 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 [Weese, 2006], but we did not include it in our experiments. We also report on some results of bwtdisk [Ferragina et al., 2012], even though it generates the BWT instead of the suffix array.

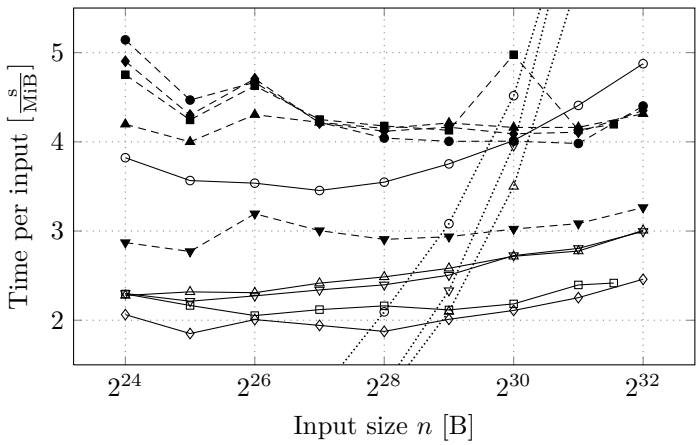
Figure 6 shows the construction time and I/O volume of eSAIS, DC3 and bwtdisk on platform A using 32-bit keys. The three algorithms eSAIS (open bullets, solid lines), DC3 (filled bullets, dashed lines), and bwtdisk (open bullets, dotted lines) were run on prefixes $T[0, 2^k)$ of all five inputs, with only Skyline being generated specifically for each size. In total the plots of eSAIS and DC3 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. The bwtdisk experiments were run only once.

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 (see Figure 7 (a)). 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.

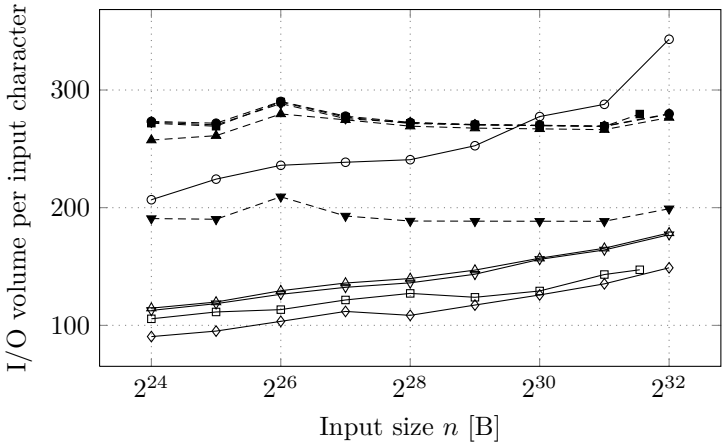
We configured bwtdisk to also use 1 GiB of main memory on platform A. Thus bwtdisk can suffix sort quite large chunks in internal memory, and behaves like its in-memory suffix sorter (divsufsort) for small input sizes. But once the input does not fit into memory, multiple chunks are merged and this merging causes the high increase in construction time seen in Figure 6. This is probably due to bwtdisk's theoretical I/O complexity, $O(n^2/(MB))$, and quadratic CPU time, $O(n^2/M)$ [Ferragina et al., 2012]. We could not measure the required I/O volume of bwtdisk, and the program does not output such statistics. The main feature

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

(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 ···○··· bwtdisk
(all inputs)

Figure 6: The two plots show (a) construction time and (b) I/O volume of eSAIS (open bullets, solid lines), DC3 (filled bullets, dashed lines), and bwtdisk (open bullets, dotted lines) on experimental platform A. The table (c) shows selected characteristics of the input strings.

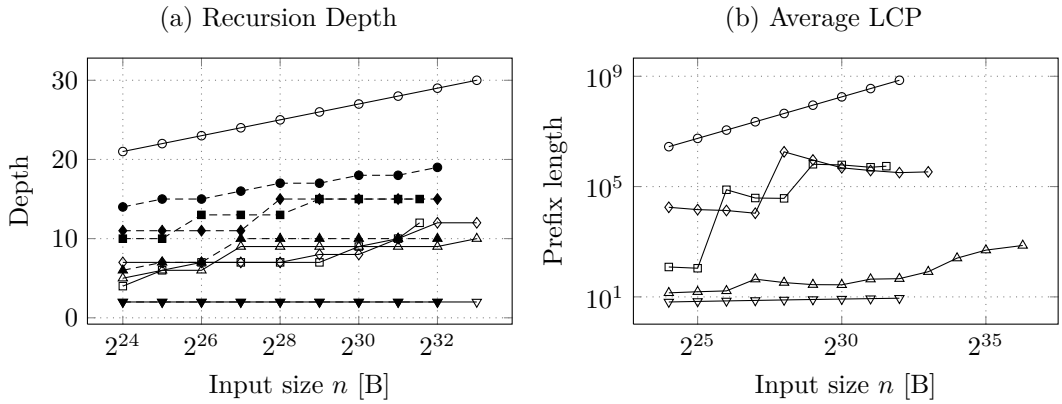


Figure 7: Plot (a) shows the maximum recursion depth reached by eSAIS and DC3 during the experiments on platform A (running with 40-bit positions). Subfigure (b) contains the average LCP of increasing 2^k slices of the inputs, calculated using eSAIS-LCP.

of bwtdisk is that it needs only very little additional disk space, which is why the authors call it "lightweight".

Besides the basic eSAIS algorithm, we also implemented a variant which "discards" sequences of multiple unique names from the reduced string prior to recursion, similar to Dementiev et al. [2008a] and Puglisi et al. [2005]. However, we discovered that this optimization has much smaller effect in eSAIS than in other suffix sorters (see Figure 8 (a)-(d)). This is probably due to the induced sorting algorithm already adapting very efficiently to the input string's characteristics.

7.2 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 [Kärkkäinen and Sanders, 2003]. Similar to Section 3.1, one must save an array LCP_N during the lexicographic naming phase. Each entry in the output LCP_T is composed of three parts: the number of equal characters found when merging sample and non-sample tuples, the expanded value from LCP_R, and the result of an RMQ on LCP_N. The second and third occur if the suffixes are ordered depending on the ranks of sub-suffixes, which is usually the case. Part one can be counted easily during merging. The second component requires processing of batched RMQs on LCP_R with the distinguishing ranks of sub-suffixes as boundaries; the result is multiplied by three for DC3. To determine the third summand,

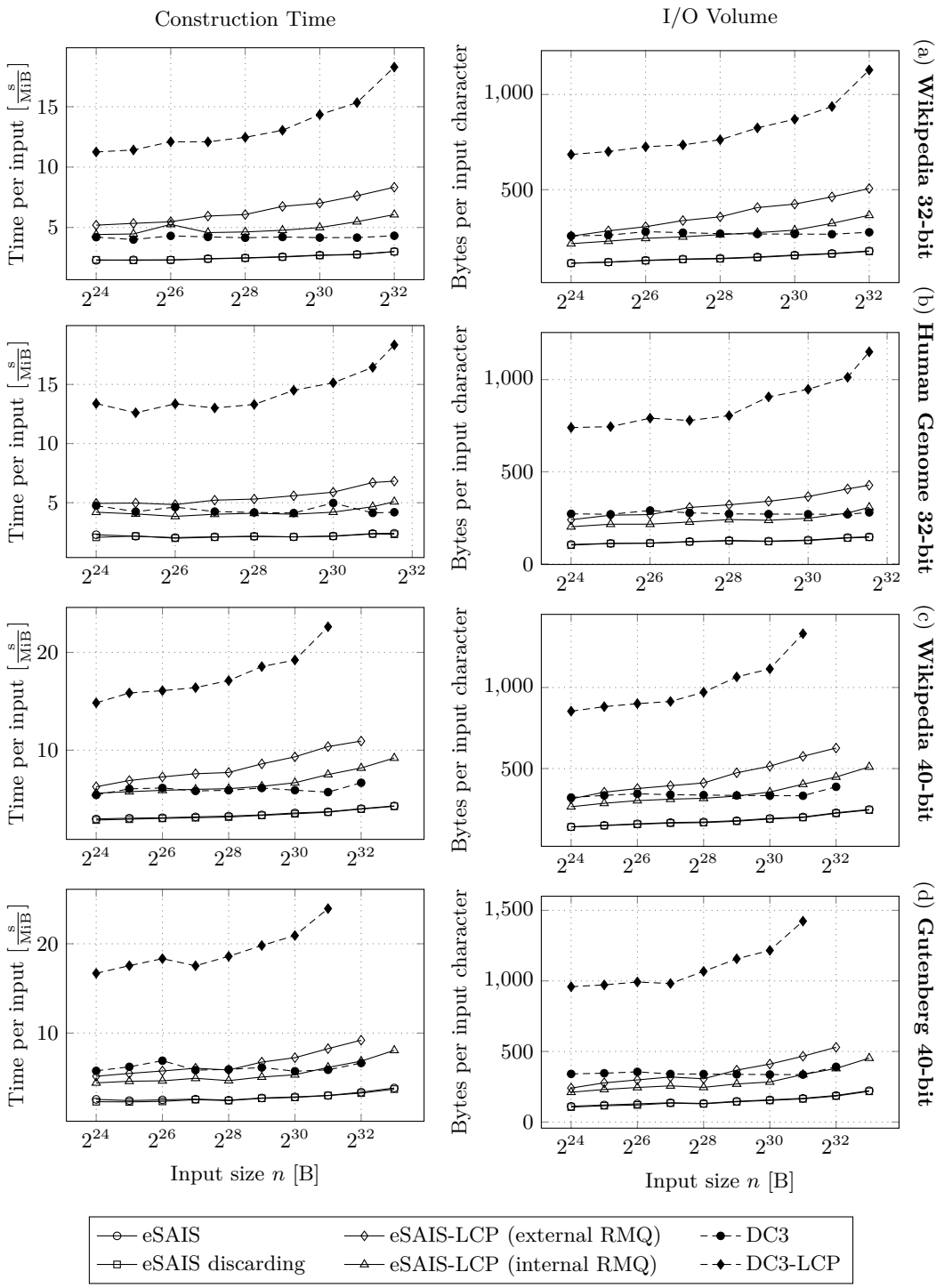


Figure 8: Subfigures (a)-(d) show construction time and I/O volume of all six implementations run on platform A for three different inputs. Subfigures (a)-(b) use 32-bit positions, while (c)-(d) runs with 40-bit.

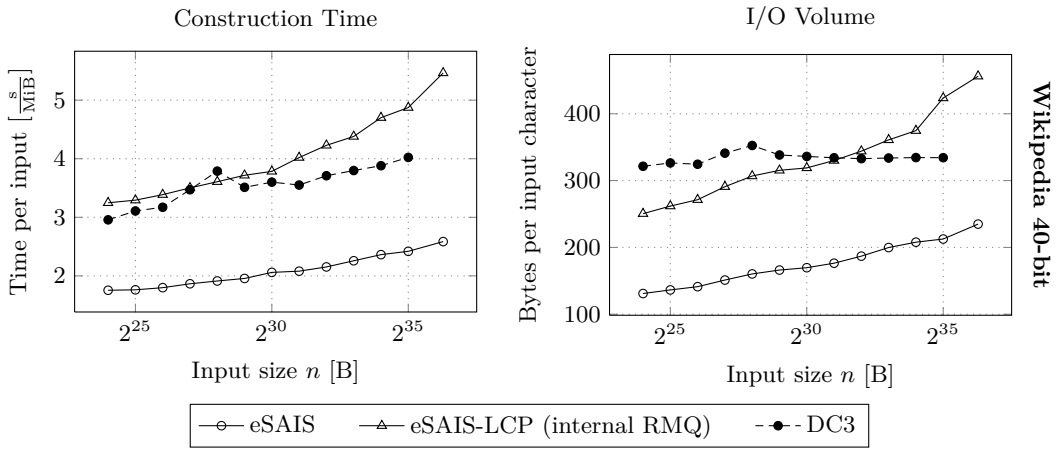


Figure 9: Measured construction time and I/O volume of three implementations is shown for the largest test instance Wikipedia run on platform B using 40-bit positions

the previously calculated value from LCP_R is used for a batched random lookup on SA_R and ISA_R (if the recursive LCP was not zero) yielding the ranks of the first pair of mismatching reduced characters. The third component represents the LCP of these character triples and is computed using an RMQ on LCP_N. These steps are similar to those needed in eSAIS-LCP (see Algorithm 3), 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 inducing process, fewer operations are required after calculating LCP_{S^*} and the RMQ ranges are "local" to the currently induced bucket.

Figure 8 (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 8 (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 9. 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.

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

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

8 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 $\mathtt{S}^*$-substrings 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 note that the final recursive stage can also output the Burrows-Wheeler transform [Burrows and Wheeler, 1994] 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

Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. 2004. Nearest Common Ancestors: A Survey and a New Algorithm for a Distributed Environment. *Theory Comput. Syst.* 37, 3 (2004), 441–456.

Antonitio, P. J. Ryan, William F. Smyth, Andrew Turpin, and Xiaoyang Yu. 2004. New suffix array algorithms — linear but not fast?. In *Proc. Fifteenth Australasian Workshop Combinatorial Algorithms (AWOCA)*. 148–156.

Lars Arge. 2003. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica* 37, 1 (2003), 1–24.

Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. 1997. On Sorting Strings in External Memory. In *Proc. STOC*. ACM Press, 540–548.

Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. 2012. LRM-Trees: Compressed Indices, Adaptive Sorting, and Compressed Permutation. *Theor. Comput. Sci.* 459 (2012), 26–41.

Marina Barsky, Ulrike Stege, and Alex Thomo. 2010. A Survey of Practical Algorithms for Suffix Tree Construction in External Memory. *Softw. Pract. Exper.* 40, 11 (2010), 965–988.

Markus J. Bauer, Anthony J. Cox, Giovanna Rosone, and Marinella Sciortino. 2012. Lightweight LCP Construction for Next-Generation Sequencing Datasets. In *Proc. WABI (LNCS)*, Vol. 7534. Springer, 326–337.

Timo Bingmann, Johannes Fischer, and Vitaly Osipov. 2013. Inducing Suffix and LCP Arrays in External Memory. In *Proc. ALENEX*. SIAM, 88–102.

Michael Burrows and David J. Wheeler. 1994. *A Block-sorting Lossless Data Compression Algorithm.* Technical Report 124. Digital Equipment Corporation.

Richard Cole and Ramesh Hariharan. 2005. Dynamic LCA Queries on Trees. *SIAM J. Comput.* 34, 4 (2005), 894–923.

Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. 2008a. Better External Memory Suffix Array Construction. *ACM J. Exp. Algorithmics* 12 (2008), Article No. 3.4.

Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008b. STXXL: Standard Template Library for XXL Data Sets. *Softw. Pract. Exper.* 38, 6 (2008), 589–637.

Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. 2008. SeqAn — An Efficient, Generic C++ Library for Sequence Analysis. *BMC Bioinformatics* 9 (2008), 11.

Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. 2000. On the sorting-complexity of suffix tree construction. *J. ACM* 47, 6 (2000), 987–1011.

Paolo Ferragina and Johannes Fischer. 2007. Suffix Arrays on Words. In *Proc. CPM (LNCS)*, Vol. 4580. Springer, 328–339.

Paolo Ferragina, Travis Gagie, and Giovanni Manzini. 2012. Lightweight Data Indexing and Compression in External Memory. *Algorithmica* 63, 3 (2012), 707–730.

Johannes Fischer. 2010. Optimal Succinctness for Range Minimum Queries. In *Proc. LATIN (LNCS)*, Vol. 6034. Springer, 158–169.

Johannes Fischer. 2011. Inducing the LCP-Array. In *Proc. WADS (LNCS)*, Vol. 6844. Springer, 374–385.

Johannes Fischer and Volker Heun. 2007. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. ESCAPE (LNCS)*, Vol. 4614. Springer, 459–470.

Johannes Fischer and Volker Heun. 2011. Space Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.* 40, 2 (2011), 465–492.

Simon Gog and Enno Ohlebusch. 2011. Fast and Lightweight LCP-Array Construction Algorithms. In *Proc. ALENEX*. SIAM Press, 25–34.

Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. 1992. New Indices for Text: PAT Trees and PAT Arrays. In *Information Retrieval: Data Structures and Algorithms*, William B. Frakes and Ricardo A. Baeza-Yates (Eds.). Prentice-Hall, Chapter 3, 66–82.

Hideo Itoh and Hozumi Tanaka. 1999. An Efficient Method for in Memory Construction of Suffix Arrays. In *Proc. SPIRE/CRIWG*. IEEE Press, 81–88.

Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. 2009. Permuted Longest-Common-Prefix Array. In *Proc. CPM (LNCS)*, Vol. 5577. Springer, 181–192.

Juha Kärkkäinen and Tommi Rantala. 2009. Engineering Radix Sort for Strings. In *Proc. SPIRE (LNCS)*, Vol. 5280. Springer, 3–14.

Juha Kärkkäinen and Peter Sanders. 2003. Simple Linear Work Suffix Array Construction. *Proc. ICALP* 2719 (2003), 943–955.

Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear Work Suffix Array Construction. *J. ACM* 53, 6 (2006), 1–19.

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. CPM (LNCS)*, Vol. 2089. Springer, 181–192.

Pang Ko and Srinivas Aluru. 2005. Space Efficient Linear Time Construction of Suffix Arrays. *J. Discrete Algorithms* 3, 2–4 (2005), 143–156.

Udi Manber and Eugene W. Myers. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* 22, 5 (1993), 935–948.

Michael A. Maniscalco and Simon J. Puglisi. 2008. An Efficient, Versatile Approach to Suffix Sorting. *ACM J. Exp. Algorithmics* 12 (2008), Article no. 1.2.

Giovanni Manzini. 2004. Two Space Saving Tricks for Linear Time LCP Array Computation. In *Proc. Scandinavian Workshop on Algorithm Theory (SWAT) (LNCS)*, Vol. 3111. Springer, 372–383.

Giovanni Manzini and Paolo Ferragina. 2004. Engineering a Lightweight Suffix Array Construction Algorithm. *Algorithmica* 40, 1 (2004), 33–50.

Ge Nong, Sen Zhang, and Wai Hong Chan. 2011. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. Computers* 60, 10 (2011), 1471–1484.

Simon J. Puglisi, William F. Smyth, and Andrew Turpin. 2005. The Performance of Linear Time Suffix Sorting Algorithms. In *Proc. Data Compression Conf. (DCC)*. IEEE Computer Society, 358–367.

Simon J. Puglisi, William F. Smyth, and Andrew Turpin. 2007. A Taxonomy of Suffix Array Construction Algorithms. *ACM Comput. Surv.* 39, 2 (2007).

Peter Sanders. 2000. Fast Priority Queues for Cached Memorys. *ACM J. Exp. Algorithmics* 5 (2000), Article No. 7.

Klaus-Bernd Schürmann and Jens Stoye. 2007. An Incomplex Algorithm for Fast Suffix Array Construction. *Softw. Pract. Exper.* 37, 3 (2007), 309–329.

Ranjan Sinha, Simon J. Puglisi, Alistair Moffat, and Andrew Turpin. 2008. Improving Suffix Array Locality for Fast Pattern Matching on Disk. In *Proc. SIGMOD*. ACM Press, 661–672.

David Weese. 2006. *Entwurf und Implementierung eines generischen Substring-Index*. Master's thesis. Humboldt University Berlin. http://www.seqan.de/publications/weeseo6.pdf.