Dismantling DivSufSort*

Johannes Fischer and Florian Kurpicz

Dept. of Computer Science, Technische Universität Dortmund, Germany
johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de

Abstract. We give the first concise description of the fastest known suffix sorting algorithm in main memory, the *DivSufSort* by Yuta Mori. We then present an extension that also computes the LCP-array, which is competive with the fastest known LCP-array construction algorithm.

Keywords: text indexing; suffix sorting; algorithm engineering

1 Introduction

The suffix array [12] is arguably one of the most interesting and versatile data structure in stringology. Despite the plethora of theoretical and practical papers on suffix sorting (see the two overview articles [18,3] for an overview up to 2007/2012), the text indexing community faces the curiosity that the fastest and most space-conscious way to construct the suffix array is by an algorithm called *DivSufSort* (coded by Yuta Mori), which has only appeared as (almost undocumented) source code, and has never been described in an academic context. The speed and its space-consciousness make DivSufSort still the method of choice in many software systems, e.g. in bioinformatics libraries[1], and in the *succinct data structures library (sdsl)* [5].

The starting point of this article was that we wanted to get a better understanding of DivSufSort's functionality and the reasons for its advantages in performance, but we could not find any arguments for this neither in the literature nor in the documentation. We therefore dove into the source code (consisting of more than 1,000 LOCs) ourselves, and want to communicate our findings in this article. We point out that just very recently Labeit et al. [10] parallelized DivSufSort, making it also the fastest *parallel* suffix array construction algorithm (on all instances but one). We think that this successful parallelization adds another reason for why a deeper study of DivSufSort is worthwile.

Our Contributions and Outline. This article pursues two goals: First, it gives a concise description of the DivSufSort-algorithm (Sect. 3), so that readers wishing to understand or modify the source code have an easy-to-use reference at hand. Second (Sect. 4), we provide and describe our own enhancement of DivSufSort that also computes related and equally important information, the array of longest common prefixes of lexicographically adjacent suffixes (*LCP-array* for short). We test our implementation empirically on a well-accepted testbed and prove it competitive with existing implementations, sometimes even little faster.

To help the reader link our description to the implementation, we show relevant excerpts from the original code[2], along with their original line numbers in the source

* This work was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

[1] https://github.com/NVlabs/nvbio, last seen 05.07.2017

[2] https://github.com/y-256/libdivsufsort, last seen 05.07.2017

Johannes Fischer, Florian Kurpicz: Dismantling DivSufSort, pp. 62–76.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathsf{T}[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
$\mathsf{type}\,(i)$	B*	A	B*	A	B*	A	B*	A	B	B*	A	A	A

Figure 1: Classification of $\mathsf{T}=$ cdcdcdcdccdd$ (our running example).

code (difsufsort.c, sssort.c, and trsort.c). In the following, we use a slanted font for variables that also appear verbatim in the source code; e.g., T for the text.

2 Preliminaries

Let $\mathsf{T}= \mathsf{T}[0]\,\mathsf{T}[1]\ldots\mathsf{T}[\mathsf{n}-1]$ be a *text* of length n consisting of characters from an ordered *alphabet* Σ of size $\sigma = |\Sigma|$. For integers $0 \le i \le j \le \mathsf{n}$, the notation $[i, j)$ represents the integers from i to $j-1$, and $\mathsf{T}[i, j)$ the *substring* $\mathsf{T}[i]\ldots\mathsf{T}[j-1]$. We call $S_i = \mathsf{T}[i, \mathsf{n})$ the i-th *suffix* of T. The *suffix array* SA of a text T of length n is a permutation of $[0, \mathsf{n})$ such that $S_{\mathsf{SA}[i]} < S_{\mathsf{SA}[i+1]}$ for all $0 \le i < \mathsf{n}-1$. In SA, all suffixes starting with the same character $\mathsf{c0} \in \Sigma$ form a contiguous interval called $\mathsf{c0}$-*bucket*. The same is true for all suffixes starting with the same two characters $\mathsf{c0}, \mathsf{c1} \in \Sigma$. We call the corresponding intervals $(\mathsf{c0}, \mathsf{c1})$-*buckets*. The *inverse suffix array* ISA is the inverse permutation of SA. The *longest common prefix* of two suffixes S_i and S_j is $\mathsf{lcp}\,(i, j) = \max\{s \ge 0\colon\ \mathsf{T}[i, i+s) = \mathsf{T}[j, j+s)\}$. The *longest common prefix array* LCP of T contains the longest common prefixes of the lexicographically consecutive suffixes, i.e., $\mathsf{LCP}[0] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}\,(\mathsf{SA}[i-1], \mathsf{SA}[i])$ for all $1 \le i \le \mathsf{n}-1$.

We classify all suffixes as follows (a technique first introduced by [7]; see Figure 1). The suffix S_i is an A-suffix (or "S_i has type A") if $\mathsf{T}[i] > \mathsf{T}[i+1]$ or $i = \mathsf{n}-1$. If $\mathsf{T}[i] < \mathsf{T}[i+1]$, then S_i is a B-suffix (or "has type B"). Last, if $\mathsf{T}[i] = \mathsf{T}[i+1]$ then S_i has the same type as S_{i+1}.[3] We further distinguish B-suffixes: if S_i has type B and S_{i+1} has type A, then suffix S_i is also a B*-suffix. Note that there are at most $\frac{\mathsf{n}}{2}$ B*-suffixes. The definition of types implies restrictions on how the suffixes are distributed within one bucket: A $(\mathsf{c0}, \mathsf{c1})$-bucket cannot contain A-suffixes if $\mathsf{c0} < \mathsf{c1}$, and it cannot contain B-suffixes if $\mathsf{c0} > \mathsf{c1}$. If $\mathsf{c0} = \mathsf{c1}$ it cannot contain B*-suffixes. The classification also induces a partial order among the suffixes (see also Fig. 2):

Lemma 1. *Let S_i and S_j be two suffixes. Then*

1. *$S_i < S_j$ if S_i has type A, S_j has type B and $\mathsf{T}[i] = \mathsf{T}[j]$, and*
2. *$S_i < S_j$ if S_i has type B*, S_j has type B but not type B* and $\mathsf{T}[i, i+1] = \mathsf{T}[j, j+1]$.*

Proof. A- and B-suffixes can only occur together in a $(\mathsf{c0}, \mathsf{c0})$-bucket. Assume that S_i and S_j start with $\mathsf{c0c0}$ followed by a (possibly empty) sequence of $\mathsf{c0}$'s and S_i, S_j have type A, B, resp. Let $u = \mathsf{T}[i + \mathsf{lcp}\,(i, j)]$ and $v = \mathsf{T}[j + \mathsf{lcp}\,(i, j)]$ be the first characters where the suffixes differ. Therefore, $u \le \mathsf{c0}$ and $v \ge \mathsf{c0}$. Since the characters differ, at least one of the inequalities is strict. The argument for the second case works analogously. □

Given two consecutive B*-suffixes S_i and S_j (i.e., there is no B*-suffix S_k such that $i < k < j$), we call the substring $\mathsf{T}[i, j+2)$ *B*-substring*. Also, for the last B*-suffix S_i (i.e., there is no B*-suffix S_k with $i < k < \mathsf{n}$), the substring $\mathsf{T}[i, \mathsf{n})$ is also called a B*-substring.

[3] This differs from [7], where S_i is always a B-suffix if $\mathsf{T}[i] = \mathsf{T}[i+1]$.

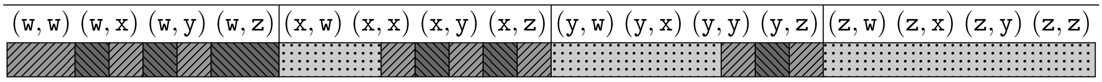

Figure 2: Position of the suffix types within the (c0, c1)-buckets for $\Sigma = \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Light gray (▒) areas represent positions of A-suffixes, gray (▨) areas represent positions of B-suffixes, and dark gray (▩) areas represent positions of B*-suffixes.

3 DivSufSort

In this section we describe DivSufSort based on its current implementation (libdivsufsort v2.0.2). The algorithm consists of three phases:

- First, we identify the types of all suffixes and compute the corresponding c0- and (c0, c1)-bucket borders. This requires one scan of the text.
- Next, we sort all B*-suffixes and place them at their correct position in SA. This is the most complicated part, as we first have to sort the B*-substrings in-place. Then, we use the ranks of the sorted B*-substrings to sort the corresponding B*-suffixes.
- In the last step, we scan SA twice to induce the correct position of all remaining suffixes. (We first scan from right to left to induce all B-suffixes, followed by a scan from left to right, inducing all A-suffixes.)

Throughout the computation we utilize two additional arrays to store information about the buckets: BUCKET_A (for A-suffixes) and BUCKET_B (for B- and B*-suffixes) of size σ and σ^2, resp. The former is used to store values associated with A-suffixes and is accessed by only one character. The latter is used to store values associated with B- and B*-suffixes and is accessed by two characters. BUCKET_B[c0, c1] is short for BUCKET_B[$|\mathsf{c0}| \cdot \sigma + |\mathsf{c1}|$] and BUCKET_BSTAR[c0, c1] is short for BUCKET_B[$|\mathsf{c1}| \cdot \sigma + |\mathsf{c0}|$], where $|\alpha|$ denotes the rank of α in the alphabet Σ. Information about both suffixes can be stored in the same array (Figure 3), as there are no B*-suffixes in (c0, c0)-buckets and no B-suffixes in (c0, c1)-buckets for c0 > c1. We denote the number of B*-suffixes by m.

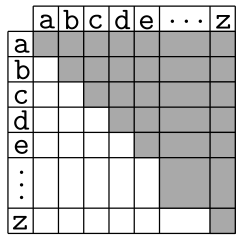

Figure 3: BUCKET_B (gray) and BUCKET_BSTAR represented as a 2-dimensional array.

3.1 Initializing DivSufSort

The initialization of DivSufSort is listed in divsufsort.c. First, we scan T from right to left (line 60), determine the type of each suffix and store the sizes of the corresponding buckets in BUCKET_A, BUCKET_B and BUCKET_BSTAR (lines 62, 69 and 65). In addition, we store the text position of each B*-suffix at the end of SA such that SA[n − m..n) contains the text positions of all B*-suffixes (line 66). We call this part of the suffix array PAb with PAb[i] = SA[n − m + i] for all $0 \leq i <$ m (line 94), see Figure 4 (a) and (b).

Next (lines 81 to 90), we compute the prefix sum of BUCKET_A and BUCKET_BSTAR, such that BUCKET_A[c0] contains the leftmost position of each c0-bucket and BUCKET_BSTAR[c0, c1] contains the rightmost position of the corresponding B*-suffixes with respect only to other B*-suffixes, i.e., the positions are in

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	0	0	0	0	0	0	0	0	0	2	4	6	9

(a)

	$	c	d	(c,c)	(c,d)
BUCKET_A	1	0	6	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(b)

	$	c							d				
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	0	0	0	0	0	0	0	0	0	2	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	**0**	**1**	**7**	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(d)

	$	c							d				
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	4	0	1	2	3	0	0	0	0	2	4	6	9

(e)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	**0**

(f)

Figure 4: SA and the buckets after the first scan of T are shown in (a) and (b). PAb (dark gray ■ in (a), (c) and (e)) contains the text positions of all B*-suffixes in *text order*. The buckets (b) contain the number of suffixes beginning with the corresponding characters. In (d), they are updated such the first position of each c0-bucket is stored in BUCKET_A[c0] (bold entires). The SA does not change during this update, see (c). In (e) we stored references to the text positions in SA[0..m $-$ 1] (light gray ■) and update the corresponding BUCKET_BSTAR with the first position in SA[0..m $-$ 1] (bold entry in (f)).

the interval $[0, \mathsf{m})$, see Figures 4 (c) and (d), where (c) remains unchanged. During the sorting step, we do not sort the text positions. Instead we sort *references* to these positions. These references are stored in SA[0..m) (line 97). During this step, BUCKET_BSTAR[c0, c1] is updated (line 97), such that it now contains the leftmost reference corresponding to a B*-suffix in the (c0, c1)-bucket within the interval $[0, \mathsf{m})$. The reference to the last B*-suffix is put at the beginning of its corresponding bucket (line 100). This reference is a special case as it has no successor in PAb that is required for the comparison of two B*-substrings, see Figure 4 (e) and (f).

3.2 Sorting the B*-Suffixes

In this section, we describe how the B*-suffixes are sorted in three steps. First, all B*-substrings are sorted independently for each (c0, c1)-bucket (lines 134 to 142) using functions defined in sssort.c. Then (second step starting at line 146), a partial ISA (named ISAb) is computed, containing the ranks of the partially sorted B*-suffixes (sorted by their initial B*-substrings). Using these ranks we compute the lexicographical order of all B*-suffixes adopting an approach similar to prefix doubling, in the last step using functions defined in trsort.c (line 159). We augment the approach with *repetition detection* as introduced by Maniscalco and Puglisi [13].

Sorting the B*-Substrings. All B*-substrings in a BUCKET_BSTAR are sorted independently and in-place. The interval of SA that has not been used yet ($\mathsf{SA}[\mathsf{m}..\mathsf{n}-\mathsf{m})$) serves as a buffer during the sorting (line 133). We refer to this part of SA as buf with $\mathsf{buf}[i] = \mathsf{SA}[\mathsf{m}+i]$ for all $0 \le i < \mathsf{n} - 2\mathsf{m}$. This part of DivSufSort can be executed in parallel by sorting the BUCKET_BSTAR in parallel, i.e., all B*-substring in one BUCKET_BSTAR are sorted sequentially, but multiple BUCKET_BSTAR are processed in parallel (see divsufsort.c, lines 105 to 131). Here, each process gets a buffer of size $\frac{|\mathsf{buf}|}{\mathsf{p}}$, where p is the number of processes. All following line numbers in this subsection refer to sssort.c.

In the default configuration we only sort 1024 elements at once (see SS_BLOCKSIZE, e.g., line 763). If the size of buf is smaller than 1024 or the size of the current bucket, the bucket is divided in smaller subbuckets which are then sorted and merged (see line 767, splitting due to the buffer size and the loop at line 770 splitting with respect to the number of elements). Lines 789 to 802 are used to merge the last considered subbuckets. If the currently sorted bucket contains the last B*-substring it is moved to the corresponding position (lines 811 and 813).

The heavy lifting is done by the function ss_mintrosort that is an implementation of *Introspective Sort* (ISS) [16]. It sorts all B*-substring within the interval [`first`, `last`] (line 310). ISS uses *Multikey Quicksort* (MKQS) [1] and *Heapsort* (HS). MKQS is used $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times to sort an interval before HS is used (if there are still elements in the interval that have been equal to the pivot each time, see line 333). MKQS divides each interval into three subintervals with respect to a pivot element. The first subinterval contains all substrings whose k-th character is smaller than the pivot, the second subinterval contains all substrings whose k-th character is equal to the pivot, and the last subinterval contains all substrings whose k-th character is greater than the pivot. We call k the depth of the current iteration (line 332). ISS is not implemented recursively; instead, a stack is used to keep track of the unsorted subintervals and the smaller subintervals are always processed first. This guarantees a maximum stack size of $\lg \ell$, where ℓ is the initial interval size [15, p. 67]. The subintervals containing the substrings whose k-th character is not equal to the pivot are sorted using MKQS $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times before using HS, where now `last` and `first` refer to the first and last positions of these intervals (lines 414 and 428).

Whenever an unsorted (sub)bucket is smaller than a threshold (8 in the default configuration), *Insertionsort* (IS) is used to sort the bucket and mark it sorted (line 326). Whenever we compare two B*-Substrings during IS, we use the function ss_compare that compares two B*-substrings starting at the current depth and compares the substrings character by character.

Throughout the sorting of the B*-substrings, substrings that cannot be fully sorted, i.e. B*-substrings that are equal, are marked by storing their bitwise negated reference (line 178). Only the first reference of such an interval is stored normally to identify the beginning of an interval of unsorted substrings (line 178). There are B*-suffixes that are not sorted completely by their initial B*-substrings e.g., in our example T= `cdcdcdcdccdd$` the B*-substring `cdcd` occurs three times – see Figure 5. Therefore, we cannot determine the order of the corresponding B*-suffixes just using their initial B*-substring. The idea of sorting the suffixes in a (c0, c1)-bucket up to a certain depth is similar to the approach of Manzini and Ferragina [14], who sort the suffixes up to a certain LCP-value.

	$	c							d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	3	0	$\tilde{1}$	$\tilde{2}$	4	0	0	0	0	2	4	6	9

(a)

Ref.	Text Pos.	B*-substring
3	6	cdcc
0	0	cdcd
1	2	cdcd
2	4	cdcd
4	9	cdd$

(b)

Figure 5: The lexicographically sorted references of the B*-substrings in $\mathsf{SA}[0..\mathsf{m}-1]$ (light gray in (a)). For readability we write $\tilde{i}$ if i is bitwise negated ($\tilde{i} < 0$ for all $0 \leq i \leq \mathsf{n}$). The content of the buckets is not changed in this step. The references, their corresponding text positions and the B*-substrings are shown in (b).

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	**-1**	0	1	2	**-1**	3	3	3	0	4	4	6	9

Figure 6: ISAb contains the inverse suffix array of the sorted B*-substrings. $\mathsf{ISAb}[i] = \mathsf{SA}[\mathsf{m}+i]$ for all $0 \leq i < \mathsf{m}$ (dark gray in SA). If $\mathsf{m} > \frac{\mathsf{n}}{3}$, ISAb overlaps with PAb. This does not matter, as we do not require the text positions at this point any more. While computing ISAb, we also mark completely sorted intervals in $\mathsf{SA}[0..\mathsf{m}-1]$. The leftmost position of a sorted interval of length ℓ is changed to -ℓ (see SA[0] and SA[4] where we store -1 as the sorted intervals contain one entry).

Computing the Partial Inverse Suffix Array. After the B*-substrings are sorted, we compute the ISA for the partially sorted B*-substrings (lines 146 to 156). The inverse suffix array for the B*-suffixes is stored in SA[m..2m) and referred to as ISAb with $\mathsf{ISAb}[i] = \mathsf{SA}[\mathsf{m}+i]$. ISAb[$i$] contains the *rank* of the i-th B*-suffix, i.e., the number of lexicographically smaller B*-suffixes. All references to line numbers in this subsection refer to divsufsort.c. We scan the SA[0..m) from right to left (line 146) and distinguish between bitwise negated references (values < 0, starting at line 154) and non-negated references (values ≥ 0, starting at line 147). In the first case, we have reached an interval where we have references of suffixes which could not be sorted comparing only the B*-substring. We assign each of those suffixes the greatest feasible rank, i.e., $\mathsf{m} - i$, where i is the number of lexicographically greater suffixes (similar to Larsson and Sadakane [11]). In addition we also store the bitwise negation of the references, i.e., the original reference. In the other case (a value ≥ 0) we simply assign the correct rank to the B*-suffix. Whenever we scan an interval of completely sorted B*-suffixes, we mark the first position of the interval in SA[0..m) with $-k$, where k is the size of the interval (line 150). Now we can identify all sorted intervals as they start with a negative value whose absolute value is the length of the interval.

In our example (see Figure 6) we have two fully sorted intervals of length 1 at SA[0] and SA[4], and an only partially sorted interval in SA[1..3].

Sorting the B*-Suffixes. In the last part of the B*-suffix sorting in DivSufSort we compute the correct ranks of all B*-suffixes and store them in ISAb. During this step, we only require information about the ranks of the suffixes and have no random access to the text, i.e., PAb is not required any more. All line numbers in this section refer to trsort.c. Using ISAb, we compute the ranks of all B*-suffixes using an approach similar to prefix doubling [11]. Instead of doubling the length of the suffixes we double the number of considered B*-substrings that can have an arbitrary length (line 563). Here, $\mathsf{ISAd}[i]$ refers to the rank of the $i+2^k$-th B*-suffix, where k is the current iteration of the doubling algorithm. Obviously, we need to update the ranks when we double the number of considered substrings, i.e., compute the new ranks for the B*-suffixes. Since the ranks in the ISA are given in text order, we can access the rank of the next (in text order) B*-substring for any given substring.

Repetition Detection. The sorting that uses the new ranks as keys is done using Quicksort (QS), which also allows us to use the *repetition* detection introduced by Maniscalco and Puglisi [13] (see line 452 for the identification and the function tr_copy for the computation of the correct ranks). A repetition in T is a substring $\mathsf{T}[i, i+rp]$ with $r \geq 2, p \geq 0$ and $i, i+rp \in [0, \mathsf{n})$ such that $\mathsf{T}[i, i+p) = \mathsf{T}[i+p, i+2p) = \cdots = \mathsf{T}[i+(r-1)p, i+rp)$. Those repetitions are a problem if S_i is a B*-suffix, since then S_{kp} is a $B^\star$-suffix for all $k \leq r$. We can simply sort all those suffixes by looking at the first character not belonging to the repetition ($\mathsf{T}[i+rp+l] \neq \mathsf{T}[i+l]$). If $\mathsf{T}[i+rp+l] < \mathsf{T}[i+l]$ then $\mathsf{T}[i+(r-1)p+1, i+rp] < \mathsf{T}[(i-1)+(r-1)p+1, (i-1)+rp]$ for all $1 < i \leq r$. The analogous case is true for $\mathsf{T}[i+rp+l] > \mathsf{T}[i+l]$, i.e., $\mathsf{T}[i+(r-1)p+1, i+rp] > \mathsf{T}[(i-1)+(r-1)p+1, (i-1)+rp]$ for all $1 < i \leq r$. This is done in lines 276 (and 282), where we increase (and decrease) the ranks of all suffixes in the repetition. The identification of a repetition is supported by QS. QS divides each interval into three subintervals (like MKQS). We chose the median rank of the B*-suffixes that are considered during this doubling step as the pivot element for QS (line 455). If the (current) rank of the first B*-suffix in the subinterval (considered in this doubling step) is equal to the pivot element, i.e., $\mathsf{ISAb}[i] = \mathsf{ISAd}[i]$ where i is the first B*-suffix in the interval, then we have found a repetition (line 452, where tr_ilg denotes the logarithm, i.e., the number of iterations until HS is used instead of QS).

Now we have computed the ISA of all B*-suffixes (stored in ISAb), i.e., we have all B*-suffixes in lexicographic order. From this point on, all line numbers refer to divsufsort.c, again. Next (see loop starting at line 162), we scan T from right to left, and when we read the i-th B*-suffix at position j, we store j at position $\mathsf{SA}[\mathsf{ISAb}[i]]$. Since we use the B*-suffixes to induce the B-suffixes (and we do not want to induce A-suffixes during the first inducing phase) we store the bitwise negation of j if S_{j-1} has type A (line 167). Figures 7a and 7b show the transition in $\mathsf{SA}[0..\mathsf{m})$ for our example. Now, $\mathsf{SA}[0..\mathsf{m})$ contains the text positions of all B*-suffixes in lexicographic order. Next (see loop beginning at line 173), we need to put these text positions at their correct position in $\mathsf{SA}[0..\mathsf{n})$ (line 182). While doing so, we update BUCKET_B and BUCKET_BSTAR such that they contain the rightmost position of the corresponding buckets (lines 177 and 185). Figures 7c and 7d show this step for our running example.

3.3 Inducing the *A*- and *B*-suffixes

Due to the types of the suffixes, we know that in any $(\mathsf{c0}, \mathsf{c1})$-bucket the A-suffixes are lexicographically smaller than the B-suffixes, and that B*-suffixes are lexicographically smaller than B-suffixes. We also know that in lexicographic order, all consecutive intervals of B-suffixes are left of at least one B*-suffix and all A-suffixes are right of at

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	-1	-4	1	0	-1	3	2	1	0	4	4	6	9

(a)

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	3	2	1	0	4	4	6	9

(b)

	$	c						d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	**1**	**6**
BUCKET_BSTAR	-	-	-	-	**1**

(d)

Figure 7: ISAb (dark gray ■ in (a) and (b)) contains the ranks of all B*-suffixes. The lexicographically sorted text positions of the B*-suffixes are shown light gray (■) in (b). Each text position i is bitwise negated if S_{i-1} has type A. In (c) all text positions of the B*-suffixes are at their correct position in SA[0..n $-$ 1] (light gray ■). The buckets (d) contain the leftmost position of the corresponding suffixes.

least one B-suffix – see Figure 2. Now we scan SA twice: once from right to left where all B-suffixes are induced (we can skip all parts of SA containing only A-suffixes), and then from left to right to induce all A-suffixes (see Figure 8 for an example of the entire inducing process). All following line numbers refer to difsufsort.c. A step-by-step example is given in Figure 8.

During the inducing of the B-suffixes, i.e., the first scan of SA (see loop starting at line 205), whenever we read an entry i in SA such that $i > 0$ (line 211), we store the entry $i-1$ at the rightmost free position (a position in which a correct text position has not been stored yet) in the (T[$i-1$], T[i])-bucket (line 220). If T[$i-2$] $>$ T[$i-1$], then S_{i-2} is an A-suffix, which is not induced during the first scan, but the bitwise negated value of $i-1$ is stored instead (line 217). Every position is overwritten with its bitwise negated value. If the position was already bitwise negated, i.e., it has been induced and the corresponding suffix has type A, it is considered during the next scan (line 226) and it is ignored otherwise. After the first traversal, all suffixes that have been used for inducing are represented by their bitwise negated position whereas all other suffixes are represented by their position, i.e., a positive integer. It should be noted that all induced suffixes are lexicographically smaller than the suffix they are induced from: if we induce from a (c0, c1)-bucket, we know that c0 $\leq$ c1, since we are considering B-suffixes. In addition, we can only induce in (c0, c1)-buckets with c1 $\leq$ c0, as only B-suffixes are considered during this traversal.

Before SA is scanned a second time, n $-$ 1 is stored at the beginning of the T[n $-$ 1]-bucket (line 234). If S_{n-2} has type A, we store n $-$ 1 (we want to induce S_{n-2} during the second scan). Otherwise, we store the bitwise negation of n $-$ 1.

During the second scan of SA (see loop starting at line 236), whenever an entry i of SA is smaller than 0 it is overwritten by its bitwise negated value, i.e., the

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	12	8	6	4	2	0	9	11	7	5	3	1	10

Figure 9: The final SA of T= cdcdcdcdccdd$.

		Scanned Interval																	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	BUCKET_A[$]	BUCKET_A[c]	BUCKET_A[d]	BUCKET_B[c, c]	BUCKET_BSTAR[c, d]	
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9	0	1	$\boxed{7}$	1	$\boxed{1}$	**First Induction Phase**
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9	0	1	7	1	1	
SA[i]	$\tilde{6}$	$\boldsymbol{\tilde{8}}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	$\boldsymbol{\tilde{9}}$	1	0	4	4	6	9	0	1	7	**0**	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	$\boldsymbol{\tilde{0}}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	$\tilde{4}$	**2**	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	**4**	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	**6**	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	**8**	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	**12**	8	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	**1**	1	7	0	1	**Second Induction Phase**
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	**11**	0	4	4	6	9	1	1	**8**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	**7**	4	4	6	9	1	1	**9**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	**5**	4	6	9	1	1	**10**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	5	**3**	6	9	1	1	**11**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	5	3	**1**	9	1	1	**12**	0	1	
SA[i]	12	8	6	4	2	**0**	$\tilde{9}$	11	7	5	3	1	9	1	1	12	0	1	
SA[i]	12	8	6	4	2	0	**9**	11	7	5	3	1	9	1	1	12	0	1	
SA[i]	12	8	6	4	2	0	9	11	7	5	3	1	**10**	1	1	**13**	0	1	

Figure 8: During the first phase, we induce B-suffixes and only scan intervals where B- and B*-suffixes occur. Each of those intervals ends left of the succeeding c0-bucket. Its borders are stored in the corresponding BUCKET_BSTAR (boxed entries, the right border is not part of the interval). After the first phase we put the last suffix at the beginning of its corresponding bucket. During the second phase we scan the whole array, as we also store the bitwise negation of all entries that have already been used for inducing. The currently considered entry is marked light gray ( ). The entries highlighted dark gray ( ) are the positions where a value is induced. The bucket that contains the position is highlighted in the same color. Entries that have changed are bold in the following row.

position of the suffix in the correct position in the suffix array (line 249). Whenever $i > 0$ (line 237) the suffix S_{i-1} is induced at the leftmost free position in the $\mathsf{T}[i-1]$-bucket (line 243). Since all remaining suffixes are induced during this scan it is sufficient to identify the border using the c0-buckets, i.e., the value stored in BUCKET_A[c0]. If the induced suffix would induce a B-suffix, its bitwise negated value is induced instead (line 240). At the end of the traversal SA contains the indices of all suffixes in lexicographic order.

4 Inducing the LCP-Array

We now show how to modify DivSufSort such that it also computes the LCP-array in addition to SA. To do so, we extend DivSufSort at three points of the computation of SA. First, we need to compute the LCP-values of all B*-suffixes. Next, during the inducing step, we also induce the LCP-values for A- and B-suffixes. For this we utilize a technique also described in [4,2] that allows us to answer RMQs on LCP using only a stack [6]. Last, we compute the LCP-values of suffixes at the border of buckets, as those values cannot be induced.

Recall that the LCP-value of two arbitrary suffixes S_i and S_j is denoted by $\mathsf{lcp}(i,j)$. We need the following additional definition: Given an array A of length ℓ and $0 \le i \le j \le \ell$, a *range minimum query* $\mathsf{RMQ}_A[i,j]$ asks for the minimum in A in the interval $[i,j]$, in symbols: $\mathsf{RMQ}_A[i,j] = \min\{A[k] : i \le k \le j\}$.

4.1 Computing the LCP-Values of the B*-Suffixes

During the sorting of the B*-suffixes (right before the B*-suffixes are put at their correct position in SA[0..n)), all lexicographically sorted B*-suffixes are in SA[0..m). There are two cases regarding m (the number of B*-suffixes). If $\mathsf{m} > \frac{\mathsf{n}}{3}$, we have overwritten the text positions of the B*-suffixes in PAb with ISAb. In this case we must compute the LCP-values naively.[4] Otherwise (we still know the text positions of all B*-suffixes), we compute their LCP-values using a sparse version of the Φ-algorithm [8], based on Observation 2, which was also used implicitly in [4,2].

Observation 2 *If $S_i, S_{i'}, S_j$ and $S_{j'}$ are B*-suffixes such that $i < i', j < j'$ and there is no other B*-suffix S_k such that $i < k < i'$ or $j < k < j'$, then* $\mathsf{lcp}(i',j') \ge \mathsf{lcp}(i,j) - (i' - i)$.

This is possible as we know the distance (in the text) of two B*-suffixes, i.e., $\mathsf{PAb}[i] - \mathsf{PAb}[j]$ is the distance of the i-th and j-th B*-suffix with $1 \le i \le j \le \mathsf{m}$. See Figure 10 for an Example. Algorithm 1 shows the *sparse* version of the Φ-algorithm. The difference to the original algorithm [8] is that the next considered suffix is an arbitrary number of character shorter than the previous one, which results in Observation 2. The computation of the LCP-values does not require any additional memory except for the n words for LCP, where we temporarily store additional data.

First (lines 1 to 4 of Algorithm 1), we fill the PHI (stored in LCP[m..2m)) such that $\mathsf{PHI}[i]$ contains the text position of the suffix that is lexicographically consecutive to the i-th suffix (text position). In $\mathsf{DELTA}[i]$ (stored in $\mathsf{LCP}[\mathsf{n}-\mathsf{m}..\mathsf{n})$) we store the text distance of the i-th and $(i+1)$-th B*-suffix (text occurrence), i.e., $\mathsf{PAb}[i+1] - \mathsf{PAb}[i]$. Then (lines 5 to 8), we compute the sparse LCP-array using Observation 2. As we store the LCP-values in PHI in text order, we need to rewrite them to LCP (line 9).

[4] For all tested instances (see Section 5) $\mathsf{m} \le \frac{\mathsf{n}}{3}$.

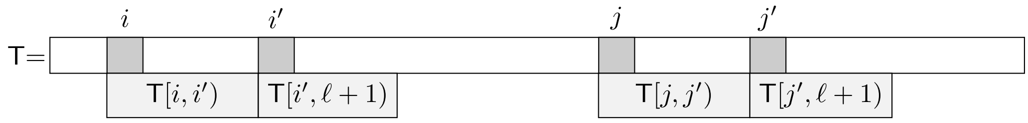

Figure 10: Let $S_i, S_j, S_{i'}$ and $S_{j'}$ be B*-suffixes such that there is no B*-suffix S_k with $i < k < i'$ or $j < k < j'$, and let the LCP-value of S_i and S_j be $\ell = \mathsf{lcp}\,(i,j) + i$. Then the LCP-value of $S_{i'}$ and $S_{j'}$ is $\mathsf{lcp}\,(i',j') = \ell - i' = \mathsf{lcp}\,(i,j) - (i' - i)$.

4.2 Inducing the LCP-Values in Addition to the SA

During the inducing of the B-suffixes, whenever a suffix is induced at position u in SA and there is already a suffix at position $u+1$ in the same $(\mathsf{c0}, \mathsf{c1})$-bucket, there are two cases:

1. The suffixes $S_{\mathsf{SA}[u]}$ and $S_{\mathsf{SA}[u+1]}$ have been induced from suffixes $S_{\mathsf{SA}[v]}, S_{\mathsf{SA}[w]}$ in the same $(\mathsf{c0}, \mathsf{c1})$-bucket; in this case $\mathsf{LCP}[u+1] = \mathsf{RMQ}_{\mathsf{LCP}}[v+1, w] + 1$.
2. Otherwise, the LCP-value is either 1 or 2, depending on the $\mathsf{c0}$-buckets $S_{\mathsf{SA}[v]}, S_{\mathsf{SA}[w]}$ are. If they are in the same bucket the LCP-value is 2 and 1 if not.

The computation of the LCP-values during the inducing of the A-suffixes works analogously. This leads to the following observation for the general case:

Observation 3 *Let* $\mathsf{SA}[u] = i, \mathsf{SA}[u+1] = j, \mathsf{SA}[v] = i+1$ *and* $\mathsf{SA}[w] = j+1$ *such that* S_i *and* S_j *are in the same* $\mathsf{c0}$*-bucket and* $u+1 < v, w$ *or* $w, v < u$*. Then* $\mathsf{LCP}[u+1] = \mathsf{RMQ}_{\mathsf{LCP}}[\min\{v,w\}+1, \max\{v,w\}] + 1$.

Not all LCP-values can be induced this way. The missing cases are covered in the next section. Instead of using a dynamic RMQ data structure, we can answer the RMQs using a *min-stack* [2,4,6]. We only need to consider RMQs for suffixes from the same $(\mathsf{c0}, \mathsf{c1})$-bucket. To this end, we build the min-stack while scanning an interval $[\mathsf{first}, \mathsf{last}]$ (from right to left) of the LCP-array. An entry on the min-stack consist of tuple $\langle k, \mathsf{LCP}[k]\rangle$. Initially, the tuple $\langle \mathsf{n}, -1\rangle$ is on the min-stack. To update the min-stack at position $i \in [\mathsf{first}, \mathsf{last}]$ we look at the top of the min-stack and remove the tuple $\langle k, \mathsf{LCP}[k]\rangle$ if $\mathsf{LCP}[k] \geq \mathsf{LCP}[i]$. We repeat this process until no tuple is removed. Then we add $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack.

Now we want to answer $\mathsf{RMQ}_{\mathsf{LCP}}[i,j]$ with $\mathsf{first} \leq i < j \leq \mathsf{last}$. (It should be noted that at this point we have not added $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack or have removed

Algorithm 1: Sparse Φ-Algorithm

Input : T, m, SA, $\mathsf{ISAb} = \mathsf{SA}[\mathsf{m}..2\mathsf{m}-1]$, $\mathsf{PAb} = \mathsf{SA}[\mathsf{n}-\mathsf{m}..\mathsf{n}-1]$ and LCP, $\mathsf{PHI} = \mathsf{LCP}[\mathsf{m}..2\mathsf{m}-1]$ $\mathsf{PHI} = \mathsf{DELTA}[\mathsf{n}-\mathsf{m}..\mathsf{n}-1]$.
Output : $\mathsf{LCP}[0..\mathsf{m}-1]$ contains the LCP-values of the B*-suffixes.

```
PHI[SA[0]] = −1
for i = 1; i ≤ m − 1; i = i + 1 do
│ PHI[SA[i]] = SA[i − 1]
│ DELTA[i − 1] = PAb[i] − PAb[i + 1]
for i = 0, p = 0; i < m; i = i + 1 do
│ while T[PAb[i] + p + 1] = T[PAb[PHI[i]] + p + 1] do
│ │ p = p + 1
│ PHI[i] = p and p = max {0, p − DELTA[i]}
for i = 0; i < m; i = j + 1 do LCP[ISAb[i]] = PHI[i];
```

any tuple from the min-stack in the process of adding it to the min-stack.) To this end, we scan the min-stack from top to bottom, until we find two consecutive tuples $\langle k, \mathsf{LCP}[k]\rangle$, $\langle k', \mathsf{LCP}[k']\rangle$ such that $k' > j$. Then, $\mathsf{RMQ}_{\mathsf{LCP}}[i,j] = \mathsf{LCP}[k]$. If we scan from left to right, the min-stack works analogously. The only difference is that the initial tuple is $\langle -1, -1\rangle$ and we search for the two consecutive tuples until $k' < j$

The min-stack is reseted whenever we arrive at a new $(\mathsf{c0}, \mathsf{c1})$-bucket, i.e., we only keep the $\langle \mathsf{n}, -1\rangle$-tuple. In the implementation, the min-stack is realized using a single array and a reference to its current top.

i	0	1	2	3	4	5	6
$\mathsf{A}[i]$	4	2	0	1	4	3	2

(a)

			$\langle 4,4\rangle$				
		$\langle 5,3\rangle$	$\langle 5,3\rangle$			$\langle 1,2\rangle$	$\langle 1,2\rangle$
	$\langle 6,2\rangle$	$\langle 6,2\rangle$	$\langle 6,2\rangle$	$\langle 3,1\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$
	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$
i	6	5	4	3	2	1	0

(b)

Figure 11: The min-stack for each *current* position i (b) while scanning A (a) from right to left. A tuple (p, v) contains the position p of the value v. For the current position i the stack can be used to answer RMQs of the type $\mathsf{RMQ}_{\mathsf{A}}[i,j]$ with $j \geq i$ by looking at elements from the top until a position k with $k \geq j$ is found.

In addition to the min-stack, we require for each $\mathsf{c0}$-bucket the position of where the last suffix has been induced from. This is the position we look for when querying the min-stack.

4.3 Special Cases during LCP Induction

There are three special cases where the LCP-value cannot be induced using the min-stack (or RMQs in general). The first case occurs if a suffix is induced next to a B*-suffix. The inducing can happen to the left or right of the already placed B*-suffix. The former case is easy as there cannot be an A- or B-suffix to the left of a B*-suffix in the same $(\mathsf{c0}, \mathsf{c1})$-bucket. Therefore, we only need to check whether the suffixes are in the same $\mathsf{c0}$-bucket to compute the LCP-value for the B*-suffix, which is either 0 or 1. The other case (a suffix is induced to the right of a B*-suffix) is more demanding, as the LCP-value must be computed. Fortunately, this can be done more sophisticated than by naive comparison of the suffixes. First, we check whether both the B*-suffix S_i and the B-suffix S_j are in the same $(\mathsf{c0}, \mathsf{c1})$-bucket. If not, the LCP-value is 1 if they occur in the same $\mathsf{c0}$-bucket, and 0 otherwise. However, if they occur in the same $(\mathsf{c0}, \mathsf{c1})$-bucket, we know that S_i has a prefix $\mathsf{c0c1}d$, $d \in \Sigma$, such that $\mathsf{c0} < \mathsf{c1} \geq d$, and that S_j has a prefix $\mathsf{c0c1}e$, $e \in \Sigma$, such that $\mathsf{c0} < \mathsf{c1} \leq e$. Hence, the LCP-value is $\max\{k \geq 0\colon\ \mathsf{T}[i+1, i+k+2) = \mathsf{T}[j+1, j+k+2)\} + 1$, i.e., the first appearance of a character not equal to $\mathsf{c1}$ in either suffix. In the last case (an A-suffix is induced next to a B-suffix) the LCP-value can be determined in an analogous way.

5 Experiments with LCP-Construction

We implemented the modified DivSufSort in C and compiled it using `gcc` version 6.2 with the compiler options -DNDEBUG, -03 and -march=native. Our implementation is available from `https://github.com/kurpicz/libdivsufsort`. We ran all experiments on a computer equipped with an Intel Core i5-4670 processor and 16 GiB RAM, using only a single core.

We evaluated our algorithm on the *Pizza & Chili* Corpus[5] and compared our implementation to the following LCP-construction algorithms (using the same compiler options): *KLAAP* [9] is the first linear-time LCP-construction algorithm. The Φ-algorithm [8] is an alternative to KLAAP that reduces cache-misses. *Inducing+SAIS* [4] is an LCP-construction algorithm (using similar ideas as in this paper) based on SAIS [17], and *naive* scans the suffix array and checks two consecutive suffixes character by character.

We also looked at LCP-construction algorithms requiring the *Burrows-Wheeler transform*, i.e., *GO* and *GO2* by Gog and Ohlebusch [6]. Since these algorithms are only available in the *succinct data structure library* (SDSL) [5], which has an emphasis on a low memory footprint, the running times are affected by that.

The results of our experiments can be found in Table 1. As a brief summary, our practical tests show that Φ (see column 1) is the fastest LCP-construction algorithm if SA is already given, while our new implementation (column 6) is faster than the only other inducing-based approach (last 2 columns).

6 Conclusions

We presented a detailed description of *DivSufSort* that has not been available albeit its wide use in different applications. We linked interesting approaches, e.g., the repetition detection, to the corresponding lines in the source code and to the original literature.

Compared with SAIS, the other popular suffix array construction algorithm based on inducing, DivSufSort is faster. We ascribe this to the two main differences between DivSufSort and SAIS: First, the sorting of the initial suffixes in SAIS (the ones that cannot be induced) is done by recursively applying the algorithm (and renaming the initial suffixes), which is slower in practice than the string-sorting and prefix doubling-like approach used by DivSufSort (which also employs techniques like repetition detection to further decrease runtime). Second, the classification of the initial suffixes differs: while the suffixes that have to be sorted initially in SAIS can be displaced during the inducing of the SA, they are not moved again in DivSufSort. This also allows DivSufSort to skip parts (containing only A-suffixes) of the SA during the first induction phase.

In addition, we showed that the LCP-array can be computed during the inducing of the suffix array in DivSufSort. This approach is faster than the previous known inducing LCP-construction algorithm based on SAIS [4], and competitive with the Φ-algorithm, i.e, the fastest pure LCP-construction algorithms.

[5] `http://pizzachili.dcc.uchile.cl/`, last seen 05.07.2017

		LCP given SA (and BWT if necessary)							SA		SA + LCP	
	Text	Φ [8]	KLAAP [9]	naive	GO [6]	GO2 [6]	inducing [this paper]	inducing [4]	DivSufSort	SAIS [17]	inducing + DivSufSort [this paper]	inducing + SAIS [4]
20 MB	dna	**0.77**	0.91	1.180	6.46	2.65	0.78	1.12	1.45	1.71	2.23	2.83
	english	**0.61**	0.77	44.72	7.90	4.03	0.64	0.91	1.45	1.65	2.09	2.56
	dblp.xml	0.54	0.55	1.640	2.56	3.92	**0.53**	0.82	1.06	1.29	1.59	2.11
	sources	**0.54**	0.57	1.530	2.87	4.26	0.57	0.85	1.07	1.41	1.64	2.26
	proteins	**0.60**	0.67	4.190	5.46	3.24	0.66	0.96	1.51	1.79	2.17	2.75
50 MB	dna	**2.02**	2.360	3.240	16.25	14.43	2.06	2.96	3.88	4.57	5.94	7.53
	english	**1.70**	2.080	65.85	15.41	12.76	1.88	2.65	3.83	4.56	5.71	7.21
	dblp.xml	1.41	1.45	4.370	9.490	9.370	**1.39**	2.17	2.93	3.53	4.32	5.70
	sources	**1.45**	1.49	6.950	10.06	10.15	1.51	2.26	2.87	3.77	4.38	6.03
	proteins	**1.77**	2.01	6.560	14.38	15.74	1.87	2.83	4.55	5.27	6.42	8.10
100 MB	dna	**4.11**	4.75	6.590	26.03	26.62	4.24	5.95	8.23	9.44	12.47	15.39
	english	**3.56**	4.28	185.9	32.57	28.09	4.02	5.62	7.96	9.49	11.98	15.11
	dblp.xml	2.85	2.89	9.040	19.91	21.49	**2.82**	4.41	6.19	7.22	9.010	11.63
	sources	**2.93**	3.02	39.85	24.92	24.46	3.07	4.62	5.98	7.72	9.050	12.34
	proteins	**3.56**	4.09	16.99	30.89	28.12	3.96	5.86	9.91	10.96	13.87	16.82
200 MB	dna	**8.25**	10.0	17.36	76.11	79.02	8.64	12.02	17.41	19.18	26.05	31.20
	english	**7.23**	8.70	1070	72.58	73.75	8.25	11.49	16.80	19.39	25.05	30.88
	dblp.xml	**5.75**	6.28	18.23	49.97	52.91	5.77	9.120	12.99	14.72	18.76	23.84
	sources	**5.98**	6.23	52.60	61.61	59.01	6.37	9.700	12.63	16.01	19.00	25.71
	proteins	**6.86**	7.94	42.60	78.78	77.40	8.33	11.82	19.73	21.65	28.06	33.47

Table 1: The first seven columns contain the times solely for the computation of LCP. Since the inducing algorithms are interleaved with the computation of SA, we subtracted the time to compute SA with the corresponding inducing approach ("inducing [this paper]" and "inducing [4]"). *GO* and *GO2* require the BWT in addition to SA; the time to compute BWT is also not included. The last two columns show the time to compute SA and LCP using the inducing approach. All times are in seconds, and are the average over 21 runs on the same input.

References

1. J. L. Bentley and R. Sedgewick: *Fast algorithms for sorting and searching strings*, in SODA, ACM/SIAM, 1997, pp. 360–369.
2. T. Bingmann, J. Fischer, and V. Osipov: *Inducing suffix and lcp arrays in external memory.*, in ALENEX, SIAM, 2013, pp. 88–102.
3. J. Dhaliwal, S. J. Puglisi, and A. Turpin: *Trends in suffix sorting: A survey of low memory algorithms*, in Proc. ACSC, Australian Computer Society, 2012, pp. 91–98.
4. J. Fischer: *Inducing the LCP-array*, in Proc. WADS, vol. 6844 of LNCS, Springer, 2011, pp. 374–385.
5. S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: Plug and play with succinct data structures*, in Proc. SEA, vol. 8504 of LNCS, Springer, 2014, pp. 326–337.
6. S. Gog and E. Ohlebusch: *Fast and lightweight LCP-array construction algorithms*, in Proc. ALENEX, SIAM, 2011, pp. 25–34.
7. H. Itoh and H. Tanaka: *An efficient method for in memory construction of suffix arrays*, in Proc. SPIRE/CRIWG, IEEE Press, 1999, pp. 81–88.
8. J. Kärkkäinen, G. Manzini, and S. J. Puglisi: *Permuted longest-common-prefix array*, in Proc. CPM, vol. 5577 of LNCS, Springer, 2009, pp. 181–192.
9. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. CPM, vol. 2089 of LNCS, Springer, 2001, pp. 181–192.
10. J. Labeit, J. Shun, and G. E. Blelloch: *Parallel lightweight wavelet tree, suffix array and fm-index construction*, in Data Compression Conference (DCC), IEEE, 2016, pp. 33–42.
11. N. J. Larsson and K. Sadakane: *Faster suffix sorting.* Theor. Comput. Sci., 387(3) 2007, pp. 258–272.
12. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches.* siam Journal on Computing, 22(5) 1993, pp. 935–948.
13. M. A. Maniscalco and S. J. Puglisi: *An efficient, versatile approach to suffix sorting.* ACM J. Experimental Algorithmics, 12 2008, p. Article no. 1.2.
14. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm.* Algorithmica, 40(1) 2004, pp. 33–50.
15. K. Mehlhorn: *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of EATCS Monographs on Theoretical Computer Science, Springer, 1984.
16. D. R. Musser: *Introspective sorting and selection algorithms.* Softw., Pract. Exper., 27(8) 1997, pp. 983–993.
17. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced-sorting*, in Proc. DCC, IEEE Press, 2009, pp. 193–202.
18. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39(2) 2007.