External Memory Generalized Suffix and LCP Arrays Construction

Felipe A. Louza[1], Guilherme P. Telles[2], and Cristina Dutra De Aguiar Ciferri[1]

[1] Institute of Mathematics and Computer Science, University of São Paulo, São Carlos, SP, Brazil
{louza,cdac}@icmc.usp.br
[2] Institute of Computing, University of Campinas, Campinas, SP, Brazil
gpt@ic.unicamp.br

Abstract. A suffix array is a data structure that, together with the LCP array, allows solving many string processing problems in a very efficient fashion. In this article we introduce eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of strings. Our algorithm relies on a combination of buffers, induced sorting and a heap. Performance tests with real DNA sequence sets of size up to 8.5 GB showed that eGSA can indeed be applied to sets of large sequences with efficient running time on a low-cost machine. Compared to the algorithm that most closely resembles eGSA purpose, eSAIS, eGSA reduced the time spent to construct the arrays by a factor of 2.5−4.8.

Keywords: generalized suffix array, generalized LCP array, external memory algorithms, text indexes, DNA indexing.

1 Introduction

Suffix arrays [1] play an important role in several string processing tasks, from pattern matching to data compression and information retrieval [2]. The suffix array combined with the longest common prefix (LCP) array provides a powerful data structure to solve many string processing problems in optimal time and space [3].

Many algorithms have been proposed for internal memory suffix arrays construction, including linear ones [4, 5]. This is also the case for LCP arrays construction [6, 7]. These algorithms are limited by internal memory size, but there is a significant number of applications that deal with a huge amount of strings that may impair the use of existing algorithms for internal memory suffix arrays construction. An example is sequence comparisons and pattern searching in molecular sequences, a field where databases have been growing at exponential rate for years. For instance, GenBank[1] has more than 150 million sequences with more than 140 billion characters.

[1] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

J. Fischer and P. Sanders (Eds.): CPM 2013, LNCS 7922, pp. 201–210, 2013.

Several algorithms have been developed for external memory suffix array construction [8–10]. More recently, Bingmann *et al.* [11] proposed eSAIS, the first external memory algorithm that constructs both suffix and LCP arrays for a single string.

Suffix arrays were also generalized to index sets of strings [12]. In the literature, works that use generalized suffix arrays do not use external memory, *e.g.* [13, 14]. On the other hand, works that investigate suffix arrays on external memory are not specifically aimed to index sets of strings. In this article we fill this gap. We introduce eGSA, an external memory algorithm to construct both generalized suffix and LCP arrays for a set of strings. We also show that eGSA can indeed be applied to sets of large sequences efficiently.

The rest of the article is organized as follows. Section 2 introduces concepts and notation, Section 3 describes the proposed algorithm, Section 4 shows performance tests that validate the algorithm, and Section 5 concludes the article and highlights future work.

2 Background

Let Σ be an ordered alphabet of symbols and let \$ be a symbol not in Σ that precedes every symbol in Σ. We denote the reflexive and transitive closure of Σ by Σ^* and the concatenation of strings or symbols by the dot operator $(\cdot)$. We define $\Sigma^{\$} = \{T \cdot \$ | T \in \Sigma^*\}$.

Let $T = T[1]T[2]\dots T[n]$ be any string of length n. A substring of T is denoted $T[i, j] = T[i]\dots T[j]$, $1 \leq i \leq j \leq n$. A prefix of T is a substring $T[1, k]$ and a suffix is a substring $T[k, n]$. We denote a suffix starting with a symbol α as an α-suffix.

A suffix array for a string T is an array of integers that provides the lexicographic order for all suffixes of T. We use the symbol $<$ for the lexicographic order relation between strings. Formally, a suffix array for a string $T \in \Sigma^{\$}$ of size n, called *SA*, is an array of integers $SA = [i_1, i_2, \dots, i_n]$ such that $T[i_1, n] < T[i_2, n] < \dots < T[i_n, n]$. The function $pos(T[k, n])$ maps the position of $T[k, n]$ in *SA*. We define an α-bucket as a block of a partition of *SA* that contains only α-suffixes.

Let $lcp(S, T)$ be the length of the longest common prefix of S and T, where $S, T \in \Sigma^{\$}$. The *LCP* array for T is an array of integers such that $LCP[i] = lcp(T[SA[i], n], T[SA[i-1], n])$ and $LCP[0] = 0$.

The *Burrows-Wheeler transform* of a string allows its efficient compression [15]. It may be stored in an array such that $BWT[i] = T[SA[i] - 1]$ if $SA[i] \neq 1$ or $BWT[i] = \$$ otherwise. *BWT* has a close relationship to *SA* and can be trivially obtained from it [10].

Suffix arrays have been generalized to index sets of strings. Given a set of k strings $\mathcal{T} = \{T_1, \dots, T_k\}$ from $\Sigma^{\$}$ with lengths $n_1, \dots, n_k$, the generalized suffix array of $\mathcal{T}$, denoted *GSA*, is an array of pairs of integers (i, j) that specifies the lexicographic order of all suffixes $T_i[j, n_i]$ of strings in $\mathcal{T}$. An order relation is defined for the tail suffixes $T_i[n_i - 1, n_i] = \$$ as $T_i[n_i - 1, n_i] < T_j[n_j - 1, n_j]$ if

$i < j$. *LCP* and *BWT* can also be generalized for sets of strings. The generalized *LCP* will be denoted *GLCP*.

3 Proposed Algorithm: eGSA

Our algorithm is called eGSA (*External Generalized Suffix and LCP Arrays Construction Algorithm*) and is based on the two-phase, multiway merge-sort presented by Garcia-Molina *et al.* [16]. The input for eGSA is a set of k strings $\mathcal{T} = \{T_1, \ldots, T_k\}$ with lengths $n_1, \ldots, n_k$ stored in the external memory and the output, which is written to external memory, is composed both by *GSA* and *GLCP* arrays for $\mathcal{T}$.

In a glance, eGSA works as follows. In the first phase it sorts the suffixes of each T_i in internal memory, obtaining SA_i, LCP_i and other auxiliary arrays, that are written to external memory. In the second phase, eGSA uses internal memory buffers to merge the previously computed arrays, obtaining *GSA* and *GLCP*. We detail each phase next.

3.1 Phase 1: Sorting

The first phase of eGSA builds SA_i and LCP_i for every $T_i \in \mathcal{T}$, using any internal memory algorithm for suffix sorting [4, 5] and *lcp* computing [6, 7]. Note that, if there is not enough internal memory available for this phase, we can use any external memory algorithm to construct them. Furthermore, two other arrays are computed, BWT_i and PRE_i, which are used to improve the second phase. The computation of all these arrays is performed in internal memory. At the end of this first phase, the arrays are written to external memory in a sequential fashion.

The prefix array for T_i, PRE_i, is defined by Barsky *et al.* [17] such that $PRE_i[j]$ is the prefix of $T_i[SA_i[j], n_i]$ of length p, for a configurable constant p. We notice that the probability that $PRE_i[j]$ is equal to $PRE_i[j+1]$ is large, since the suffixes are sorted in SA_i. Then, to avoid redundancy, we adopt a different strategy, similar to the lef-justified approach in [18], and construct PRE_i through non-overlapping substrings as $PRE_i[j] = T_i[SA_i[j] + h_j, SA_i[j] + h_j + p]$, where $h_j = min(LCP_i[j], h_{j-1} + p)$ and $h_0 = 0$.

Figure 1 shows the output structures of this phase of eGSA for $T_1 = GATAGA\$$ and $p = 3$. The last column is merely illustrative and shows the suffixes $T_1[SA_1[j], n_1]$. For simplicity when $SA_1[j] + h_j + p > n_1 = 7$ we consider $T[SA_1[j] + h_j + p] = \$$.

3.2 Phase 2: Merging

The second phase of eGSA merges the arrays computed in the first phase to obtain *GSA* and *GLCP* for $\mathcal{T}$, as follows.

Let $R_i = \langle SA_i, LCP_i, BWT_i, PRE_i \rangle$. Each R_i is partitioned into r_i blocks $R_i^1, \ldots, R_i^{r_i}$, having b consecutive elements from each array except perhaps for

j	$SA_1[j]$	$LCP_1[j]$	$BWT_1[j]$	$PRE_1[j]$	$T_1[SA[j], n_1]$
1	6	-	A	\$\$\$	**\$**
2	5	0	G	A\$\$	**A\$**
3	3	1	T	GA\$	**AGA\$**
4	1	1	G	TAG	**ATAG**A\$
5	4	0	A	GA\$	**GA\$**
6	0	2	\$	TAG	**GATAG**A\$
7	2	0	A	TAG	**TAG**A\$

Fig. 1. Output structures of phase 1, for $T_1 = GATAGA\$$ and $p = 3$

$R_i^{r_i}$. For each R_i the algorithm uses two internal memory buffers: a string buffer S_i, which stores substrings up to s symbols of T_i, and a partition buffer B_i, which stores a block R_i^j. $B_i[j]$ is composed of $\langle SA_i[k], LCP_i[k], BWT_i[k], PRE_i[k]\rangle$, for $j = k \bmod b$. We also use two other buffers. The output buffer stores d elements from the *GSA* and *GLCP* arrays. The induced buffer has size c and stores information that is necessary in the inducing strategy, to be discussed below. The values of s, b, d and c determine the amount of internal memory used in this phase.

Each block R_i^1 is initially loaded into its buffer B_i. Then the heading elements of each buffer B_i are inserted into a binary heap. The smallest suffix in the heap is moved to the output buffer and replaced by the next element from the same buffer B_i. This operation is repeated until all partition blocks are empty. When the output buffer is full, it is written to external memory.

The most sensitive operation in this phase is the comparison of elements from each buffer. Using a naïve approach may require too many random disk accesses. This is due the fact that for every SA_i involved in the comparison, the corresponding suffixes must be accessed in external memory, loaded into string buffers and then compared.

To reduce disk accesses, we propose an enhanced comparison method composed of three strategies: (i) prefix assembling; (ii) *lcp* comparisons; and (iii) inducing suffixes. These strategies are described below.

Prefix Assembly. Let j be the index of the smallest element in the buffer B_i. PRE_i is used to load the initial prefix of $T_i[SA_i[j], n_i]$ into S_i with no disk accesses, just concatenating previous $PRE_i[k]$, for $k = 1, 2, \ldots, j$. As j changes, buffer S_i is updated such that $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$, where $h_j = min(LCP_i[j], h_{j-1} + p)$, $h_0 = 0$, and $\#$ is an end-of-buffer marker not in Σ. Thus, if a string comparison does not involve more than $h_j + p$ symbols, a disk access is not necessary. Otherwise $\#$ is reached and T_i is accessed in the external memory.

The last column of Figure 1 illustrates in bold the prefixes recovered by prefix assembly. For instance, if $j = 5$ then $h_5 = 0$ and S_1 stores $GA\$$. Next, when $j = 6$ then $h_6 = min(LCP_i[6], h_5 + p) = min(2, 0 + 3) = 2$, and $S_1[3, 3 + 3 - 1] = S_1[3, 5]$ receives $PRE_i[5] = TAG$. In this case, $S_1 = S_1[1, 2] \cdot S_1[3, 5] \cdot \# = GA \cdot TAG \cdot \# = GATAG\#$.

LCP Comparisons. Let X, Y and Z be nodes in the binary heap storing $B_a[i]$, $B_b[j]$ and $B_c[k]$, respectively. Suppose that node X is the parent of Y and Z. As $X < Y$ and $X < Z$, then $T_a[SA_a[i], n_a] < T_b[SA_b[j], n_b]$ and $T_a[SA_a[i], n_a] < T_c[SA_c[k], n_c]$. The *lcp* values can be used to speed up suffix comparisons in the heap [19]. The following lemma formalizes this relation. The proof is simple and will be omitted.

Lemma 1. *Let S_1, S_2 and S_3 be strings, such that $S_1 < S_2$ and $S_1 < S_3$. If $lcp(S_1, S_2) > lcp(S_1, S_3)$ then $S_2 < S_3$. If $lcp(S_1, S_2) < lcp(S_1, S_3)$ then $S_2 > S_3$. Otherwise, if $lcp(S_1, S_2) = lcp(S_1, S_3) = l$ then $lcp(S_2, S_3) \geq l$.*

The order of Y and Z can be determined using Lemma 1, and if $lcp(X, Y) = lcp(X, Z)$ then $lcp(Y, Z) \geq lcp(X, Y) = l$, and Y and Z can be compared directly starting from position l. As X is removed from the heap, $B_a[i]$ is moved to the output buffer and X is replaced by another node W storing $B_a[i+1]$. The order of W with respect to its children can also be determined by Lemma 1 along the heap, as the right position for W is searched. The *lcp* values in the heap are updated as nodes are swapped. Hence, using *lcp* values many direct comparisons of strings that are in the external memory are avoided.

Inducing Suffixes. *Induced sorting* is the determination of the order of unsorted suffixes from already sorted suffixes that is used by many internal memory algorithms [4]. We apply an induced sorting approach based on the following lemma, whose proof is straightforward.

Lemma 2. *Let Suff be the set of all suffixes of $\mathcal{T}$, $T_i \in \mathcal{T}$, $1 \leq j \leq n_i$ and $\alpha \in \Sigma$. If $T_i[j, n_i]$ is the smallest element of Suff (w.r.t. the lexicographic order) then $T_i[j-1, n_i] = \alpha \cdot T_i[j, n_i]$ is the smallest α-suffix of Suff $\setminus \{T_i[j, n_i]\}$.*

Lemma 2 can be used to sort the suffixes of T_i as follows. *Suff* starts with every suffix of T_i and as the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ is found, $T_i[j, n_i]$ is removed from *Suff* and inserted into the smallest available position of the α-bucket. Then $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ is induced to the smallest available position in the β-bucket, $\alpha, \beta \in \Sigma$.

Note that if $\alpha > \beta$ the suffix $T_i[j-1, n_i]$ was already sorted. Moreover, the induced suffixes $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ cannot be removed from *Suff* because they must induce suffixes $T_i[j-2, n_i]$ as well. To this end, when the smallest β-suffix $T_i[j-1, n_i]$ is the smallest suffix in *Suff*, the β-bucket is read starting from the second element. As the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from *Suff*. Also, if $\alpha = \beta$, then reading induced suffixes from the β-bucket can cause the induction of already induced suffixes. So no induction is done when $\alpha \geq \beta$.

However, this approach is not efficient to sort a single string T_i, since it is always necessary to find the smallest suffix $T_i[j, n_i]$. But in a merge algorithm, the smallest suffix is one of those remaining in buffer B_i, and can be determined efficiently using the heap. Let *Suff* be the set of all suffixes of $\mathcal{T}$ and suppose that $B_i[k]$ is at the root of the heap. Then $T_i[j, n_i]$ is the smallest suffix in *Suff*,

and using the approach described previously we can induce $T_i[j-1, n_i]$ if $\alpha < \beta$. For this, we use BWT_i to determine if $T_i[j] < T_i[j-1]$ and whether or not $T_i[j-1, n_i]$ can be induced.

The smallest suffixes are moved to the output buffer, and the induced suffixes are written to the induced buffer, which is written to external memory as it gets full. When the smallest β-suffix $T_i[j-1, n_i]$ is the smallest in *Suff*, the β-bucket is read from external memory, and induces other suffixes as necessary. Note that there is no need to compare the induced suffixes in this step, it is sufficient only to follow the order imposed by the β-bucket in the heap.

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap. Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix. Then $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1$. But since the suffixes $T_a[i, n_a]$ and $T_b[j, n_b]$ may not be consecutive in *GSA*, the value of $LCP(T_a[i, n_a], T_b[j, n_b])$ may not be obtained directly. For that, let the range minimum query on *GLCP* be $rmq(i, j) = \min_{i \leq k \leq j}\{GLCP[k]\}$. Since $T_a[i, n_a]$ and $T_b[j, n_b]$ are already sorted, $LCP(T_a[j, n_a], T_b[j, n_b]) = rmq(pos(T_a[j, n_a] + 1, pos(T_b[j, n_b]))$. The *rmq* values may be computed as *GLCP* is moved to the output buffer storing the *min* function for each $\alpha \in \Sigma^*$.

Therefore, when a suffix $T_i[j, n_i]$ is induced in the second phase, its corresponding *LCP* is also induced from the *rmq* values. As induced suffixes may also induce, the corresponding *LCP* must be stored in the induced buffer together with the induced suffixes in their α-bucket. As the induced suffixes are recovered from the external memory, the *LCP* must also be recovered to update *rmq*.

4 Performance Evaluation

The performance of eGSA was analyzed through tests with real DNA sequences from the genomes of (1) Human, (2) Medaka, (3) Zebrafish, (4) Cow, (5) Mouse and (6) Chicken, which were obtained from the Ensembl genome database[2]. We generated 5 datasets, described in Table 1. We preprocessed these datasets to remove the character N (unknown). Each character in a dataset uses one byte. The mean and maximum LCP values provide an approximation of suffix sorting difficulty [9].

Our algorithm was implemented in ANSI/C. The construction of the suffix and *LCP* arrays in the first phase of eGSA was performed by the *inducing+saislite* algorithm [6], which uses approximately $9 \times |T_i|$ bytes. We used $p = 23$ for the size of PRE_i. The buffers S_i, B_i, output and induced were set to use 200 KB, 10 MB, 64 MB and 16 MB of internal memory, respectively. We remark that eGSA uses 1 byte for each character in S_i. The output produced by eGSA was validated using a trivial checking algorithm. The source code is freely available from `http://code.google.com/p/egsa/`.

[2] `http://www.ensembl.org/`

Table 1. Datasets used in our experiments. Column 2 indicates the genomes that compose each dataset. Column 3 shows the number of strings (i.e. chromosomes). Columns 4 and 5 show the computed mean and maximum *lcp* values. Column 6 reports the dataset size.

Dataset	Genomes	Number of strings	mean LCP	max. LCP	Input size (GB)
1	2	24	19	2,573	0.54
2	6	30	17	5,476	0.92
3	3, 6	56	58	71,314	2.18
4	2, 3, 4	80	44	71,314	4.26
5	1, 4, 5, 6	105	59	168,246	8.50

We compared our algorithm with the eSAIS algorithm [11], which is the fastest algorithm to date that computes both suffix and LCP arrays in external memory. However, eSAIS is aimed at indexing only one string T_i. To use this algorithm to index a set of strings, we concatenated all strings in $\mathcal{T}$, replacing \$ of each T_i by a new terminal symbol $\$_i$, such that $\$_i < \$_j$ if $i < j$ and $\$_i < \alpha$ for each $\alpha \in \Sigma$. This approach limits the number k of strings that can be indexed. For DNA sequences, using 1 byte for each character, k is limited by $256 - |\{A, C, G, T\}| = 252$. We are aware of the existence of the algorithms by Bauer *et al.* [20, 21] that aim at indexing sets of fixed size, small strings in external memory. However, we did not consider comparing them with eGSA because they solve a different problem.

The eGSA was compiled by GNU gcc compiler, version 4.6.3, with optimizing option -O3. The experiments were conducted in the Linux Ubuntu 12.04/64 bits operating system, running on an Intel Core i7 2.67 GHz processor 8MB L2 cache, 12 GB of internal memory and a 1 TB SATA hard disk with 5900 RPM and 64MB cache. The amount of internal memory usage across the experiments was restricted to 4 GB.

Table 2 shows the experimental results of eGSA and eSAIS execution. Although the comparison is not totally fair because eSAIS was not designed for multiple strings, eGSA have consistently outperformed eSAIS by a factor of 2.5−4.8 in time (columns μs/input byte). Then we may safely conclude that eGSA is an efficient algorithm for generalized suffix and *LCP* arrays construction on external memory. Moreover, phase 2 of eGSA used only 1.1 GB of internal memory for dataset 5.

In the same fashion, we can analyze eGSA through efficiency, that is the proportion of time for which the CPU busy, not waiting for I/O. As shown in Table 2, the efficiency decreases for dataset 5, while still more efficient than eSAIS. This is not caused only by the dataset size but also by the maximum *lcp* value for the dataset. We can see that the CPU time ratio for both algorithms on datasets 4 and 5 is close, what indicates that I/O is probably related to efficiency loss. We believe that adjusting buffer sizes may improve efficiency, and that, in particular, the string buffer size s is more closely related to this issue. As with many other external memory algorithms, buffer size adjust is often necessary.

Table 2. Results for the comparison of eGSA and eSAIS. Columns 2 and 3 report the running time in microseconds per input byte. Columns 4 and 5 report the total running time (wallclock) in seconds. Columns 6 and 7 report the total cputime time not accounting for the time of I/O. Columns 8 and 9 report the efficiency of each algorithm, that is the proportion of cputime by wallclock. Finally, column 10 reports the ratio of eSAIS cputime by eGSA cputime.

Dataset	μs/input byte		wallclock (sec)		cputime (sec)		efficiency		cputime ratio
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
1	5.86	1.72	3,413	1,005	1,236	687	0.36	0.68	1.80
2	5.97	1.24	5,883	1,228	2,110	715	0.36	0.58	2.95
3	6.23	2.27	14,596	5,314	4,385	3,349	0.30	0.63	1.31
4	6.41	2.31	29,383	10,590	8,542	7,566	0.29	0.71	1.13
5	7.24	2.79	66,106	25,502	16,652	13,003	0.25	0.51	1.28

Furthermore, we also registered that the proportion of induced suffixes is 37.4% on the average, what shows that inducing suffixes is a major improvement strategy in eGSA.

The theoretical cost of phase 1 of eGSA is dominated by the algorithms used to construct SA_i and LCP_i. In phase 2, the number of node swaps in the heap is bounded by $N \log k$, where N is the sum of the k string lengths. Each node swap requires comparing a number of characters that is equal to the maximum value of *lcp* for $\mathcal{T}$ (*maxlcp*). The cost of this phase is dominated by the $(N \log k)maxlcp$ comparisons, beyond I/O operations.

5 Conclusions and Future Work

In this article we proposed eGSA, which is the first external memory algorithm to construct both generalized suffix and LCP arrays for a set of strings. The proposed algorithm was validated through performance tests using real DNA sequences from different species, which were combined in datasets with different number of strings and data volume.

The results showed that eGSA is efficient. Compared to the eSAIS algorithm, the algorithm that most closely resembles eGSA purpose, eGSA reduced the time spent to construct the arrays by a factor of 2.5−4.8.

Another advantage of eGSA is that it may be employed to build generalized suffix and LCP arrays from suffix and LCP arrays that have already been computed individually for strings in a dataset. Moreover, eGSA may be used to construct the core data structures used by LOF-SA search algorithms [18] and to build generalized suffix trees in external memory [17]. Furthermore, it may be applied to construct the Longest Previous Factor array, which is used in text compression and for detecting motifs and repeats [22].

We are currently extending the eGSA algorithm to also construct a generalized *Burrows-Wheeler transform* of a set of strings. Another future work is redesigning the algorithm for multiple disks, one for write operations and the others for read operations. The data structures and algorithms used in our approach suggest

that the running time of eGSA is subquadratic, but a remaining task is to formalize the asymptotic analysis both for memory and I/O operations and to compare them with experimental results for this and other datasets.

Acknowledgments. This work has been supported by the Brazilian agencies FAPESP, CNPq and CAPES.

References

1. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
2. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York (1997)
3. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
4. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
5. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. Data Compression Conference, pp. 193–202 (2009)
6. Fischer, J.: Inducing the LCP-array. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 374–385. Springer, Heidelberg (2011)
7. Gog, S., Ohlebusch, E.: Fast and lightweight lcp-array construction algorithms. In: Proc. Meeting on Algorithm Engineering & Experiments, pp. 25–34 (2011)
8. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica 32(1), 1–35 (2002)
9. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM J. of Experimental Algorithmics 12 (2008)
10. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)
11. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and lcp arrays in external memory. In: Proc. Meeting on Algorithm Engineering & Experiments, pp. 88–103 (2013)
12. Shi, F.: Suffix arrays for multiple strings: A method for on-line multiple string searches. In: Jaffar, J., Yap, R.H.C. (eds.) ASIAN 1996. LNCS, vol. 1179, pp. 11–22. Springer, Heidelberg (1996)
13. Pinho, A., Ferreira, P., Garcia, S., Rodrigues, J.: On finding minimal absent words. BMC bioinformatics 10, 137 (2009)
14. Arnold, M., Ohlebusch, E.: Linear time algorithms for generalizations of the longest common substring problem. Algorithmica 60(4), 806–818 (2011)
15. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Systems Research (1994)
16. Garcia-Molina, H., Widom, J., Ullman, J.D.: Database System Implementation. Prentice-Hall, Inc., Upper Saddle River (1999)
17. Barsky, M., Stege, U., Thomo, A., Upton, C.: A new method for indexing genomes using on-disk suffix trees. Proc. ACM International Conference on Information and Knowledge Management 236(1-2), 649 (2008)
18. Sinha, R., Puglisi, S.J., Moffat, A., Turpin, A.: Improving suffix array locality for fast pattern matching on disk. Proc. ACM SIGMOD, 661–672 (2008)

19. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. Information Processing Society of Japan Digital Courier 4, 69–78 (2008)
20. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the bwt of string collections. Theoretical Computer Science (2012) (in press)
21. Bauer, M.J., Cox, A.J., Rosone, G., Sciortino, M.: Lightweight LCP Construction for Next-Generation Sequencing Datasets. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 326–337. Springer, Heidelberg (2012)
22. Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Wale, T.: Computing the longest previous factor. European J. of Combinatorics 34(1), 15–26 (2013)