Engineering a Lightweight External Memory Suffix Array Construction Algorithm*

Juha Kärkkäinen, Dominik Kempa

Department of Computer Science, University of Helsinki, Finland
{Juha.Karkkainen | Dominik.Kempa}@cs.helsinki.fi

Abstract

We describe an external memory suffix array construction algorithm based on constructing suffix arrays for blocks of text and merging them into the full suffix array. The basic idea goes back over 20 years and there has been a couple of later improvements, but we describe several further improvements that make the algorithm much faster. In particular, we reduce the I/O volume of the algorithm by a factor $\mathcal{O}(\log_\sigma n)$. Our experiments show that the algorithm is the fastest suffix array construction algorithm when the size of the text is within a factor of about five from the size of the RAM in either direction, which is a common situation in practice.

1 Introduction

The suffix array [12, 9], a lexicographically sorted array of the suffixes of a text, is the most important data structure in modern string processing. It is the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [14]. Modern text books spend dozens of pages in describing applications of suffix arrays, see e.g. [16]. In many of the applications, the construction of the suffix array is the main bottleneck in space and time, even though a great effort has gone into developing better algorithms [17].

For internal memory, there exists an essentially optimal suffix array construction algorithm (SACA) that runs in linear time using little extra space in addition to what is needed for the input text and the output suffix array [15]. However, little *extra* space is not good enough for large text collections such as web crawls, Wikipedia

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

* Supported by the Academy of Finland grant 118653 (ALGODAN).

or genomic databases, which may be too big for holding even the text alone in RAM. There are also external memory SACAs that are theoretically optimal with internal work $\mathcal{O}\Big(n \log_{M/B}(n/B)\Big)$ and I/O complexity $\mathcal{O}\Big((n/B) \log_{M/B}(n/B)\Big)$ [11, 4], where M is the size of the RAM and B is the disk block size. Furthermore, they are practical and have implementations [7, 4] that are fully scalable in the sense that they are not seriously limited by the size of the RAM. However, constant factors in practical running time and disk space usage are significant. The currently best implementation, eSAIS [4], needs $28n$ bytes of disk space, which is probably the main factor limiting its practical scalability.

In this paper, we focus on an algorithm, which we call SAscan, that lies between internal memory SACAs and eSAIS in scalability. SAscan is a true external memory algorithm in the sense that it can handle texts that do not fit in internal memory, but its time complexity $\Omega(n^2/M)$ makes it hopelessly slow when the text is much larger than the RAM. However, when the text is too large for an internal memory SACA, i.e., larger than about one fifth of the RAM size, but not too much bigger than the RAM, SAscan is probably the fastest SACA in practice. SAscan is also lightweight in the sense that it uses less than half of the disk space of eSAIS, and can be implemented to use little more than what is needed for the text and the suffix array.

The basic approach of SAscan was developed already in the early days of suffix arrays by Gonnet, Baeza-Yates and Snider [9]. The idea is to partition the text into blocks that are small enough so that the suffix array for each block can be constructed in RAM. The block suffix arrays are then merged into the full suffix array. After constructing the suffix array for each block, the algorithm scans the previously processed part of the text and determines how each suffix of the scanned text compares to the suffixes of the current block. The information collected during the scan is then used for performing the merging.

The early version of SAscan depended heavily on the text not having long repeats and thus had a poor worst case time complexity. This problem was solved by Crauser and Ferragina [6], who developed an improved version with worst case time complexity $\mathcal{O}\big((n^2/M) \log M\big)$ and I/O complexity of $\mathcal{O}\big(n^2/(MB)\big)$. The algorithm was further improved by Ferragina, Gagie and Manzini [8], who reduced the time complexity to $\mathcal{O}\big((n^2/M) \log \sigma\big)$, where σ is the size of the text alphabet, and the disk space usage to little more than what is needed for the input and the output.

In this paper, we describe several further improvements to the SAscan algorithm. The first improvement is a new merging technique that reduces the I/O complexity of SAscan to $\mathcal{O}\big(n^2/(MB \log_\sigma n) + n/B\big)$ and provides a substantial speedup in practice too. Another target of improvement is the rank data structure that plays a key role in the algorithm. In theory, we observe that the time complexity can be reduced to $\mathcal{O}\big((n^2/M) \log(2 + (\log \sigma / \log \log n))\big)$ by plugging in the rank data structure of Belazzougui and Navarro [3]. In practice, we improve the rank data structure used in the implementation by Ferragina, Gagie and Manzini by applying alphabet partitioning [2] and fixed block boosting [10]. Finally, we reduce the size of the internal memory data structures by more than one third, which allows the algorithm to use bigger and fewer blocks improving the running time significantly.

We show experimentally that our practical improvements reduce the running time of SAscan by more than a factor of three. We also show that the algorithm is faster than eSAIS when the text size is less than about six times the available RAM, at which point the disk space usage of eSAIS is already well over 150 times the available RAM.

2 Preliminaries

Let $\mathsf{X} = \mathsf{X}[0..m)$ be a string over an integer alphabet $[0..\sigma)$. For $i = 0, \ldots, m-1$ we write $\mathsf{X}[i..m)$ to denote the *suffix* of X of length $m-i$, that is $\mathsf{X}[i..m) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[m-1]$. Similarly, we write $\mathsf{X}[i..j)$ to denote the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j-1]$ of length $j-i$. If $i = j$, the substring $\mathsf{X}[i..j)$ is the empty string, also denoted by ε.

The suffix array $\mathsf{SA_X}$ of X contains the starting positions of the non-empty suffixes of X in the lexicographical order, i.e., it is an array $\mathsf{SA_X}[0..m)$ which contains a permutation of the integers $[0..m)$ such that $\mathsf{X}[\mathsf{SA_X}[0]..m) < \mathsf{X}[\mathsf{SA_X}[1]..m) < \cdots < \mathsf{X}[\mathsf{SA_X}[m-1]..m)$.

The partial suffix array $\mathsf{SA_{X:Y}}$ is the lexicographical ordering of the suffixes of XY with a starting position in X, i.e., it is an array $\mathsf{SA_{X:Y}}[0..m)$ that contains a permutation of the integers $[0..m)$ such that $\mathsf{X}[\mathsf{SA_{X:Y}}[0]..m)\mathsf{Y} < \mathsf{X}[\mathsf{SA_{X:Y}}[1]..m)\mathsf{Y} < \cdots < \mathsf{X}[\mathsf{SA_{X:Y}}[m-1]..m)\mathsf{Y}$. Note that $\mathsf{SA_{X:\varepsilon}} = \mathsf{SA_X}$ and that $\mathsf{SA_{X:Y}}$ is usually similar but not identical to $\mathsf{SA_X}$. Also, $\mathsf{SA_{X:Y}}$ can be obtained from $\mathsf{SA_{XY}}$ by removing all entries that are larger or equal to m.

3 Overview of the Algorithm

Let a string $\mathsf{T}[0..n)$ be the text. It is divided into blocks of size (at most) m, where m is chosen so that all the in-memory data structures of $\mathcal{O}(m \log n)$ bits fit in the RAM. The blocks are processed starting from the end of the text. Assume that so far we have processed $\mathsf{Y} = \mathsf{T}[i..n)$ and constructed the suffix array $\mathsf{SA_Y}$. Next we will construct the partial suffix array $\mathsf{SA_{X:Y}}$ for the block $\mathsf{X} = \mathsf{T}[i-m..i)$ and merge it with $\mathsf{SA_Y}$ to form $\mathsf{SA_{XY}}$.

The suffixes in $\mathsf{SA_{X:Y}}$ and $\mathsf{SA_Y}$ are in the same relative order as in $\mathsf{SA_{XY}}$ and we just need to know how to merge them. For this purpose, we compute the *gap array* $\mathsf{gap_{X:Y}}[0..m]$, where $\mathsf{gap_{X:Y}}[i]$ is the number of suffixes of Y that are lexicographically between the suffixes $\mathsf{SA_{X:Y}}[i-1]$ and $\mathsf{SA_{X:Y}}[i]$ of XY. Formally, for $i \in [1..m)$,

$$\begin{aligned}
\mathsf{gap_{X:Y}}[0] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{Y}[j..|\mathsf{Y}|) < \mathsf{X}[\mathsf{SA_{X:Y}}[0]..m)\mathsf{Y}\}| \\
\mathsf{gap_{X:Y}}[i] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{X}[\mathsf{SA_{X:Y}}[i-1]..m)\mathsf{Y} < \mathsf{Y}[j..|\mathsf{Y}|) < \mathsf{X}[\mathsf{SA_{X:Y}}[i]..m)\mathsf{Y}\}| \\
\mathsf{gap_{X:Y}}[m] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{X}[\mathsf{SA_{X:Y}}[m-1]..m)\mathsf{Y} < \mathsf{Y}[j..|\mathsf{Y}|)\}| \,.
\end{aligned}$$

The construction of $\mathsf{gap_{X:Y}}$ scans Y and is the computational bottleneck of the algorithm.

The merging of $\mathsf{SA_{X:Y}}$ and $\mathsf{SA_Y}$ is trivial with the help of $\mathsf{gap_{X:Y}}$ but involves a lot of I/O for reading $\mathsf{SA_Y}$ and writing $\mathsf{SA_{XY}}$. The total I/O volume is $\mathcal{O}\big(n^2/m\big)$ in units of $\mathcal{O}(\log n)$-bit words. However, we can reduce the I/O by delaying the

merging. We write $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ to disk and then proceed to process the next block. Once all partial suffix arrays and gap arrays have been computed, we perform one multiway merging of the partial suffix arrays with the help of the gap arrays. The I/O volume for merging is reduced to $\mathcal{O}(n)$ words.

Suppose that during the construction of $\mathsf{SA}_{\mathsf{X:Y}}$ or $\mathsf{gap}_{\mathsf{X:Y}}$ we need to compare two suffixes of $\mathsf{SA}_{\mathsf{XY}}$, at least one of which begins in X. In the worst case, the suffixes could have a very long common prefix, much longer than m, making it impossible to perform the comparison without a lot of I/O — unless we have extra information about the order of the suffixes. In our case, that extra information is provided by a bitvector gt_{Y}, which tells whether each suffix of Y is lexicographically smaller or larger than Y itself. Formally, for all $i \in [0..|\mathsf{Y}|)$,

$$\mathsf{gt}_{\mathsf{Y}}[i] = \begin{cases} 1 & \text{if } \mathsf{Y}[i..|\mathsf{Y}|) > \mathsf{Y} \\ 0 & \text{if } \mathsf{Y}[i..|\mathsf{Y}|) \leq \mathsf{Y} \end{cases}$$

With the help of gt_{Y}, two suffixes of XY, at least one of which begins in X, can be compared in $\mathcal{O}(m)$ time. The algorithms for constructing $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ perform more complex operations than plain comparisons, but they use the same bitvector to avoid extra I/Os resulting from long common prefixes.

In summary, for each text block X we perform the following steps:

1. Given X, $\mathsf{Y}[0..m)$ and $\mathsf{gt}_{\mathsf{Y}}[0..m)$, compute $\mathsf{SA}_{\mathsf{X:Y}}$.
2. Given X, $\mathsf{SA}_{\mathsf{X:Y}}$, Y and gt_{Y}, compute $\mathsf{gap}_{\mathsf{X:Y}}$ and $\mathsf{gt}_{\mathsf{XY}}$.

The output bitvector $\mathsf{gt}_{\mathsf{XY}}$ is needed as input for the next block. The two other arrays $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ are stored on disk until all blocks have been processed. The final phase of the algorithm takes all the partial suffix arrays and gap arrays as input and produces the full suffix array SA_{T}.

4 Details and Analysis

The first stage in processing a block X is constructing the partial suffix array $\mathsf{SA}_{\mathsf{X:Y}}$. In the full paper, we show how to construct a string Z such that $\mathsf{SA}_{\mathsf{Z}} = \mathsf{SA}_{\mathsf{X:Y}}$. We can then construct the suffix array using any standard SACA; In the implementation we use Yuta Mori's divsufsort [13].

The partial Burrows–Wheeler transform [5] of X is an array $\mathsf{BWT}_{\mathsf{X:Y}}[0..m)$ defined by:

$$\mathsf{BWT}_{\mathsf{X:Y}}[i] = \begin{cases} \mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[i] - 1] & \text{if } \mathsf{SA}_{\mathsf{X:Y}}[i] > 0 \\ \$ & \text{if } \mathsf{SA}_{\mathsf{X:Y}}[i] = 0 \end{cases},$$

where \$ is a special symbol that does not appear in the text. For a character c and an integer $i \in [0..m]$, the answer to the rank query $\mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:Y}}}(c, i)$ is the number of occurrences of c in $\mathsf{BWT}_{\mathsf{X:Y}}[0..i)$. Rank queries can be answered in $\mathcal{O}(\log(2 + (\log \sigma / \log\log n)))$ time using a linear space data structure [3]. In practice, we use a simpler data structure, described in the full paper, that requires $4.125m$ bytes of space.

For a string S, let $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{S})$ be the number of suffixes of XY starting in X that are lexicographically smaller than S. Let $\mathsf{C}[0..\sigma)$ be an array, where $\mathsf{C}[c]$ is the number of positions $i \in [0..m)$ such that $\mathsf{X}[i] < c$. In the full paper, we prove the following lemma.

Lemma 4.1 *Let* $k = \mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{S})$ *for a string* S*. For any symbol* c*,*

$$\mathsf{sufrank}_{\mathsf{X:Y}}(c\mathsf{S}) = \mathsf{C}[c] + \mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:Y}}}(c,k) + \begin{cases} 1 & \text{if } \mathsf{X}[m-1] = c \text{ and } \mathsf{Y} < \mathsf{S} \\ 0 & \text{otherwise} \end{cases}.$$

Note that when $\mathsf{S} = \mathsf{Y}[j..|\mathsf{Y}|)$, we can replace the comparison $\mathsf{Y} < \mathsf{S}$ with $\mathsf{gt}_{\mathsf{Y}}[j] = 1$. Thus, given $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j..|\mathsf{Y}|))$, we can easily compute $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j-1..|\mathsf{Y}|))$ using the lemma, and we only need to access $\mathsf{Y}[j-1]$ in Y and $\mathsf{gt}_{\mathsf{Y}}[j]$ in gt_{Y}. Hence, we can compute $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j..|\mathsf{Y}|))$ for $j = |\mathsf{Y}|-1, \ldots, 0$ with a single sequential pass over Y and gt_{Y}. This is all that is needed to compute $\mathsf{gap}_{\mathsf{X:Y}}$ and $\mathsf{gt}_{\mathsf{XY}}$, which are the output of the second stage of processing X.

The final phase of the algorithm is merging the partial suffix arrays into the full suffix array SA_{T}. For $k \in [0..\lceil n/m \rceil)$, let $\mathsf{X}_k = \mathsf{T}[km..(k+1)m)$, $\mathsf{Y}_k = \mathsf{T}[(k+1)m..n)$, $\mathsf{SA}_k = \mathsf{SA}_{\mathsf{X}_k:\mathsf{Y}_k}$ and $\mathsf{gap}_k = \mathsf{gap}_{\mathsf{X}_k:\mathsf{Y}_k}$. The algorithm shown below moves suffixes from the input suffix arrays to the output suffix array in ascending lexicographical order.

```
1: for k = 0 to ⌈n/m⌉ − 1 do i_k ← 0
2: for i = 0 to n − 1 do
3:     k ← 0
4:     while gap_k[i_k] > 0 do
5:         gap_k[i_k] ← gap_k[i_k] − 1
6:         k ← k + 1
7:     SA_T[i] ← SA_k[i_k] + km
8:     i_k ← i_k + 1
```

The correctness of the algorithm is based on the following invariants maintained by the algorithm: (1) i_k is the number of suffixes already moved from SA_k to SA_{T}, and (2) $\mathsf{gap}_k[i_k]$ is the number of suffixes remaining in SA_{k+1}, SA_{k+2}, ..., $\mathsf{SA}_{\lceil n/m \rceil - 1}$ that are smaller than $\mathsf{SA}_k[i_k]$.

Theorem 4.2 *SAscan can be implemented to construct the suffix array of a text of length* n *over an alphabet of size* σ *in* $\mathcal{O}\left(\frac{n^2}{M}\log\left(2+\frac{\log\sigma}{\log\log n}\right)\right)$ *time and* $\mathcal{O}\left(\frac{n^2\log\sigma}{MB\log n} + \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ *I/Os in the standard external memory model (see [18]) with RAM size* M *and disk block size* B*, both measured in units of* $\Theta(\log n)$*-bit words. Under the reasonable assumption that* $M \geq B\log_\sigma n$*, the I/O complexity is* $\mathcal{O}\left(\frac{n}{B}\left(1+\frac{n\log\sigma}{M\log n}\right)\right)$.

In the full paper, we describe an implementation that needs $5.2m$ bytes of RAM and at most $11.5n$ bytes of disk space, and could be implemented to use just $6.5n$ bytes of disk space.

5 Experimental Results

We performed experiments on a machine with a 3.16GHz Intel Core 2 Duo CPU with 6144KiB L2 cache running Linux (Ubuntu 12.04, 64bit, kernel 3.2). All

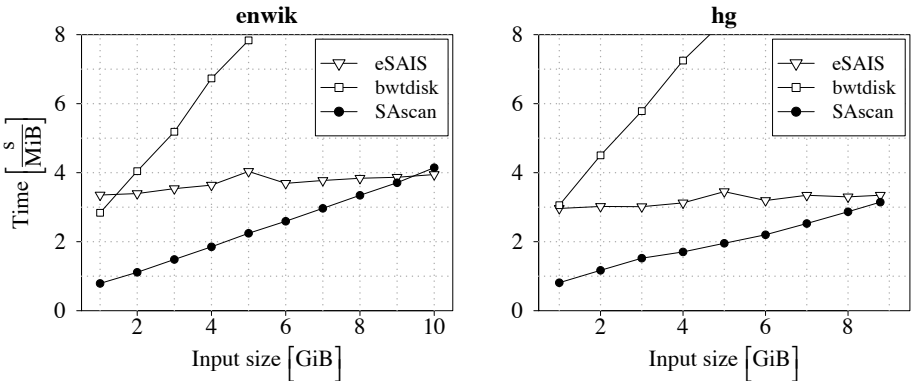

Figure 1: Scalability of SAscan compared to eSAIS and bwtdisk.

programs were compiled using g++ version 4.6.4 with -O3 -DNDEBUG options. To reduce the time to run the experiments, we artificially restricted the RAM size to 2GiB using the Linux boot option mem, and the algorithms were allowed to use at most 1.5GiB. We used two big test files: a concatenation of three different Human genomes[1,2,3] (hg) and a prefix of the English Wikipedia dump[4] (enwik).

The first experiment measures the scalability of our new algorithm. We computed the suffix array for varying length prefixes of each testfile using our algorithm and compared to eSAIS – currently the fastest algorithm for building SA in external memory. In addition, we also show the runtimes of bwtdisk[5], which is essentially the Ferragina–Gagie–Manzini version of SAscan though it constructs the BWT instead of the suffix array. The suffix array version of bwtdisk would be slower as it needs more I/O during merging. The results are given in Figure 1. SAscan is faster than eSAIS up to input sizes of about 9GiB, which is about six times the size of the RAM available to the algorithms. It is this ratio between the input size and the RAM size that primarily determines the relative speeds of SAscan and eSAIS. Note that the main limitation to the scalability of eSAIS is the disk space requirement, which is about 170 times the available RAM at the crossing point. The runtime of bwtdisk is always at least 3.5 times the runtime of SAscan, showing the dramatic effect of our improvements. In the second experiment, we take a closer look at the effect of our improvements on the runtime. More precisely, we show a detailed runtime breakdown after turning on individual improvements one by one: the fast merging of partial suffix arrays, the space-efficient representation of the gap array (which reduces the RAM usage from $8m$ to $5.2m$ bytes), and the optimized rank data structure. The results are presented in Figure 2. Each of the improvements produces a significant speedup. The combined effect is more than a factor of three.

[1] http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/

[2] ftp://public.genomics.org.cn/BGI/yanhuang/fa/

[3] ftp.ncbi.nlm.nih.gov/genbank/genomes/Eukaryotes/vertebrates_mammals/Homo_sapiens/

[4] http://dumps.wikimedia.org/enwiki/

[5] http://people.unipmn.it/manzini/bwtdisk/

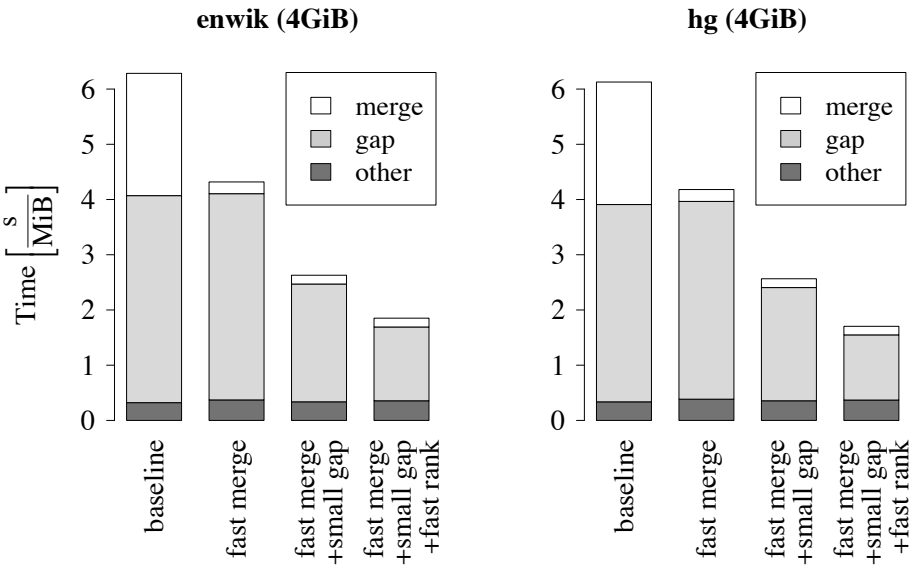

Figure 2: Effects of various optimizations on runtime. We separated the runtime into three components: merging suffix arrays, gap array construction and other ($\mathcal{O}(n)$ time) computations.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. ISAAC*, pages 315–326, 2010.

[3] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. ESA*, pages 181–192, 2012.

[4] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proc. ALENEX*, pages 103–112, 2013.

[5] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[6] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[7] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM J. Experimental Algorithmics*, 12:Article 3.4, 2008.

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

[9] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.

[10] J. Kärkkäinen and S. J. Puglisi. Fixed-block compression boosting in FM-indexes. In *Proc. SPIRE*, pages 174–184, 2011.

[11] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

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

[13] Y. Mori. libdivsufsort, a C library for suffix array construction. `http://code.google.com/p/libdivsufsort/`.

[14] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[15] G. Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):Article 15, 2013.

[16] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[17] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.

[18] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.