Faster Lightweight Suffix Array Construction

Michael A. Maniscalco* Simon J. Puglisi†

Abstract

The suffix array is a data structure formed by sorting the suffixes of a string into lexicographical order. It is important for a variety of applications, perhaps most notably pattern matching, pattern discovery and block-sorting data compression. The last decade has seen intensive research toward efficient construction of suffix arrays with algorithms striving not only to be fast, but also "lightweight" (in the sense that they use small working memory).

In this paper we describe a new lightweight suffix array construction algorithm. By exploiting several interesting properties of suffixes in combination with cache concious programming we acheive excellent runtimes. Extensive experiments show our approach to be faster that all other known algorithms for the task.

1 Introduction

The suffix array is a data structure formed by sorting the suffixes of a string into lexicographical order. Each suffix is represented by an integer denoting its starting position in the string, and so, on usual architectures the entire suffix array requires $4n$ bytes of storage, where n is the length of the string.

Suffix arrays were proposed by Manber and Myers [11] originally as a space-economical alternative to the suffix tree [22, 14, 21] — a data structure whose "myriad virtues" were well known for many string processing problems [2]. Recently, in [1], it was shown how the suffix array, when augmented with relatively small auxilliary information, could be used to simulate any algorithm on the suffix tree in the same asymptotic time bounds (and usually much faster in practice). Suffix sorting (the process of constructing the SA) is also of relevance to lossless data compression, being the most computationally difficult step in the *Burrows-Wheeler transform* (BWT)[5]. Even with straightforward coding techniques, a string that has undergone the BWT can be compressed very close to entropy. More recently, the relationship between the SA and the BWT has been explored by several authors, and has given rise to the field of *compressed full-text indexing* [10].

Due to the interesting and useful applications of SAs outlined above, the last decade has seen intensive research in the development of efficient suffix array construction algorithms (SACAs), with a recent survey paper counting a dozen distinct approaches [18]. SACAs strive not only to be fast, but also "lightweight" (in the sense that they use small working memory).

*Independant Researcher; `www.michael-maniscalco.com`; Email: `michael@michael-maniscalco.com`

†Department of Computing, Curtin University of Technology, GPO Box U1987, Perth WA 6845, Australia; Email: `puglissj@computing.edu.au`

Since 2003, SACAs have existed that operate in $\Theta(n)$ time [7, 9, 8] but they are not (yet) practical, and require much more time and space than many supra-linear approaches [17]. The current state of the art in *practical* SACAs is algorithm MF by Manzini and Ferragina [13]. Via a range of (at times quite complex) heuristics, MF achieves fast runtimes on realworld inputs, and uses little more than $5n$ bytes of working space. The drawback of MF is its $O(n^2 \log n)$ time bound, which leads to unacceptable performance for some classes of strings [17].

In this paper we describe a new approach to suffix sorting. By exploiting several interesting properties of suffixes in combination with cache concious programming we arrive at an implementation which is faster than MF (and other leading SACAs) and uses almost the same amount of working memory. Furthermore, we show that our algorithm outperforms MF on the classes of strings which cause MF difficulty.

The remainder of this document is as follows: §2 sets notation and definitions used throughout, we then introduce our algorithm in §3. Experimental results are given in §4 before conclusions and directions for future work are offered in §5.

2 Definitions and Notation

Let Σ be a *constant*, *indexed* alphabet consisting of symbols σ_j, $j = 1, 2, \ldots, |\Sigma|$ ordered $\sigma_1, \sigma_2, \cdots, \sigma_{|\Sigma|}$. In this paper we will assume the common case that $|\Sigma| \in 0..255$, where each symbol requires one byte of storage. We refer to pair of consecutive symbols as a *bigram*. Throughout we assume Σ is minimal, that is, $\Sigma = \{0, 1, 2, \ldots, |\Sigma| - 1\}$.

Throughout we consider a finite, nonempty string $x = x[0..n] = x[0]x[1] \ldots x[n]$ of $n + 1$ symbols. The first n symbols of x are drawn from Σ and comprise the actual input. The final character $x[n]$ is a special "end of string" character, \$, defined to be lexicographically smaller than any other character in Σ. We assume that $n < 2^{32}$, implying that an integer in the range $0..n$ can be stored in 4 bytes. For $i = 0, \ldots, n$ we write $x[i..n]$ to denote the suffix of x of length $n - i + 1$, that is $x[i..n] = x[i]x[i + 1] \cdots x[n]$. For simplicity we will frequently refer to suffix $x[i..n]$ simply as "suffix i".

We are interested in computing the *suffix array* of x, which we write SA_x or just SA. The suffix array is an array SA$[0..n]$ which contains a permutation of the integers $0..n$ such that $x[\mathrm{SA}[0]..n] < x[\mathrm{SA}[1]..n] < \cdots < x[\mathrm{SA}[n]..n]$. Put another way $\mathrm{SA}[j] = i$ iff $x[i..n]$ is the j^{th} suffix of x in ascending lexicographical order (*lexorder*). We also define the inverse suffix array ISA$[0..n]$ such that $\mathrm{ISA}[i] = j$ iff $\mathrm{SA}[j] = i$. Thus, ISA$[i]$ provides the lexicographic rank of $x[i..n]$ in constant time. We say an array of strings is in k-order if the strings are in lexorder according to their length $k \geq 1$ prefixes.

3 New Algorithm

This section has four parts. The first describes a way to identify a specific set of suffixes that, once sorted, enable the entire SA to be produced in an efficient way. In the second and third parts, we introduce an inventive memory arrangement which allows us to recast a powerful technique from [12] to improve cache behaviour; finally we describe the means by which our algorithm handles strings that have very high average lcp (longest common prefix).

3.1 Choosing a sample of suffixes to sort

Central to our algorithm is a sampling method that allows us to sort only a small *sample* of the total suffixes, and later induce the order of the others (the *non-sample*) in linear time. In this section we explain how the sample is selected and how to order the non-sample once the sample is sorted. We delay until §3.3-§3.5 a description of how the sample itself is sorted. Throughout this section we use the notation S_X to denote the set of sample suffixes chosen by method X.

Itoh and Tanaka, with there aptly named SACA *twostage*, were the first to make use of the sampling idea [6]. They divide suffixes into two sets U_{IT} and V_{IT} defined as follows

$$\begin{aligned} U_{IT} &= \{x[i..n] \colon x[i] < x[i+1]\} \\ V_{IT} &= \{x[i..n] \colon x[i] \geq x[i+1]\} \end{aligned}$$

For example, consider the string $x = edabdccdeedab$, annotated below to show suffixes belonging to each set U_{IT} and V_{IT}:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
type$_{IT}$	V	V	U	U	V	V	U	U	V	V	V	U	V	$-$

Only the suffixes in V_{IT} need to be sorted, that is $S_{IT} = V_{IT}$. The members of U_{IT} can then be sorted and merged in time proportional to their number to form the whole suffix array (exactly how this is done will be explained shortly). This sampling idea has been subsequently refined by other authors [9, 15]. Ko and Aluru define sets U_{KA} and V_{KA}

$$\begin{aligned} U_{KA} &= \{x[i..n] \colon x[i..n] < x[i+1..n]\} \\ V_{KA} &= \{x[i..n] \colon x[i..n] > x[i+1..n]\} \end{aligned}$$

and then choose to sort the smaller of the two sets, so $S_{KA} = \min_{||}(U_{KA}, V_{KA})$, and the number of suffixes requiring explicit sorting is limited to at most $n/2$. This idea is that of Itoh and Tanaka applied at a "suffix-level" rather than a "character-level" and so $V_{KA} \supseteq V_{IT}$, as illustrated by our example:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
type$_{KA}$	V	V	U	U	V	V	U	U	V	V	V	U	V	$-$

In [15], Mori takes matters a step further by showing how only a subset S_M of the suffixes in S_{KA} need to be sorted. Assume $S_{KA} = U_{KA}$.

$$S_M = \{x[i..n] \colon x[i..n] \in U_{KA} \text{ and } x[i+1..n] \in V_{KA}\}$$

From the order of the suffixes in S_M, the order of those in S_{KA} can be induced after which point the algorithm procedes the same as that of Ko and Aluru. Below we show the contents S_M (below that of KA) for our running example, with an '$*$' indicating inclusion in S_M.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
type$_{KA}$	V	V	U	U	V	V	U	U	V	V	V	U	V	$-$
S_M	$-$	$-$	$-$	$*$	$-$	$-$	$-$	$*$	$-$	$-$	$-$	$*$	$-$	$-$

Table 1: The percentage of suffixes requiring explicit sorting under various sampling schemes for files in the MF-corpus.

String	IT	KA	M
sprot34	55	36	29
rfc	59	42	26
howto	54	36	29
reuters	51	35	30
linux	57	41	27
jdk13c	52	35	30
etext99	49	34	31
chr22	63	41	27
gcc	57	40	27
w3c2	52	36	30

It is easy to see that, in general, $|S_M| \leq |S_{KA}| \leq |S_{IT}|$ where S_X denotes the sample selected by method X [1]. However, in practice we have found a marked difference between the schemes. Table 1 shows the size of the sample for each scheme for the data files in the MF-corpus [2].

We now turn our attention to how, once the suffixes in S_M have been sorted, the entire suffix array can be produced. In our example, $S_M = \{3, 7, 11\}$. Assume these suffixes are in their final place in SA, as shown below

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SA	(−)	11	(−)	(−)	3	(−)	(−)	(−	−)	(−)	7	(−	−)	(−)
group	$	ab		b$	bd	cc	cd	da		dc	de	ed		ee

The parentheses () indicate *group* boundaries, such that all suffixes belonging to the same group share a common prefix of length 2 [3]. Now, to sort the type U_{KA} suffixes, we scan SA in its current state from right to left. For each suffix (ie. non-empty location) SA[i] that we encounter in the scan, if SA[i] − 1 is of type U_{KA} we place SA[i] − 1 at the current end of group x[SA[i] − 1]x[SA[i]] and decrement the end of that group. When the scan is complete, all the members of U_{KA} are in their final place in SA.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	Description
SA	(−)	11	(−)	(−)	3	(−)	(−)	(−	−)	(−)	7	(−	−)	(−)	Initial
SA	(−)	11	(−)	(−)	3	(−)	6	(−	−)	(−)	7	(−	−)	(−)	SA[10]=7, 6 ∈ U_{KA}, place 6
SA	(−)	11	(−)	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[6]=6, 5 ∈ U_{KA}, place 5
SA	(−)	11	(−)	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[5]=5, 4 ∉ U_{KA}, no action
SA	(−)	11	2	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[4]=3, 2 ∈ U_{KA}, place 2
SA	(−)	11	2	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[2]=2, 1 ∉ U_{KA}, no action
SA	(−)	11	2	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[1]=11, 10 ∉ U_{KA}, no action
group	$	ab		b$	bd	cc	cd	da		dc	de	ed		ee	

[1] Consider the string a^n\$ to see the case where $|S_M| = |S_{KA}|$.

[2] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`

[3] Group boundaries are easily computed from bigram frequencies.

To complete the suffix array we scan SA from left to right operating in a similar way that algorithms IT [6] and KA [9] do: for each suffix SA[i] encountered in the scan, if suffix SA[i] -1 is of type V we place SA[i] -1 at the current head of group x[SA[i] -1]x[SA[i]] and increment the head of that group by one. There is one caveat: before we begin the scan we must place the terminating suffix (that begins with \$) in SA[0]. The salient steps of this final scan are illustrated below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	Description
SA	13	(−)	2	(−)	3	5	6	(−	−)	(−)	7	(−	−)	(−)	Initial, after placing 13 in SA[0]
SA	13	11	2	12	3	5	6	(−	−)	(−)	7	(−	−)	(−)	SA[0]=13, 12 $\in V_{KA}$, place 12
SA	13	11	2	12	3	5	6	10	(−)	(−)	7	(−	−)	(−)	SA[1]=11, 10 $\in V_{KA}$, place 10
SA	13	11	2	12	3	5	6	10	1	(−)	7	(−	−)	(−)	SA[2]=2, 1 $\in V_{KA}$, place 1
SA	13	11	2	12	3	5	6	10	1	(−)	7	9	0	(−)	SA[3]=12, 11 $\notin V_{KA}$, no action
...															
SA	13	11	2	12	3	5	6	10	1	4	7	9	0	8	SA[11]=9, 8 $\in V_{KA}$, place 8
group	\$	ab		b\$	bd	cc	cd	da		dc	de	ed		ee	

In the above discussion we have assumed $U_{KA} < V_{KA}$. When the opposite is true, the algorithm to sort and merge all the suffixes is symmetric to the one described above. Another solution is to reverse the order of the alphabet at the very beginning (and recode the string) and then reverse the sorted order (that is, the order in which suffixes occur in the suffix array) in a single pass at the end. Finally, it is important to note that this sampling method is only suitable for applications in which $|\Sigma|^2$ is a manageable size — fortunately this is most often the case.

3.2 Input Transformation and Memory Organization

The input string $x[0..n]$ is mapped onto an array of integers ISA'$[0..n]$. The mapping is done via a function code(i) that transforms the bigram $x[i..i+1]$ and stores the result at ISA'[i] — code(i) is computed with the aid of several small arrays.

- Array bi_freq[0..65536] where bi_freq[b] is the number of times bigram b[0..1] occurs in x (ie. its frequency).
- Arrays lo_rank[0..65536] and hi_rank[0..65536] defined inductively as follows:
  - lo_rank[0] $= 0$;
  - hi_rank[b] $=$ lo_rank[b] $+$ bi_freq[b]; and
  - lo_rank[b] $= k$ where k is the smallest integer such that $k >$ high_rank[$b-1$] and $k \mod 256 = 0$.

The code function is now simply defined code(i) $=$ hi_rank[$x[i..i+1]$]. The transformation has the following important properties:

Observation 1. *The lexorder of the suffixes of ISA' is the same as for x.*

Observation 2. *For any bigram b in x we have $b = c \mod 256$ for $c \in$ lo_rank[b]..hi_rank[b].*

The first quality means sorting suffixes of ISA' is equivalent to sorting those of x. The second is vital to us efficiently recovering x from the contents of ISA' at a later stage. In coming sections we show how sorting suffixes of ISA' rather than x allows the sorting process to be drastically sped up.

The sample suffixes of ISA' (equivalently x) are stored in an array SP$[0..n/2]$ of $n/2+1$ pointers, initially in 2-order. The 2-order is easily acheived via a counting sort, using the bifreq array defined above. Usually $|S| < n/2$ so SP is not always full. In fact, we only need to allocate space for $\min(n/4+1, |S|)$ pointers in SP. We require SP to have size at least $n+1$ bytes so that later the space can be reused to reproduce and store x, the original string.

Summarizing memory usage, ISA' requires $4n$ bytes and SP requires 1 or $2n$ bytes, depending on the size of the sample set, S. Thus our total memory usage is 5 or $6n$ bytes, save for several small arrays of size 65536 bytes. The only other memory required is for the recursion stack of a quicksort procedure, which we go to pains to keep small (in fact constant) [4].

3.3 Primary Sorting Procedure

The suffixes in SP are sorted using a variation of Multi-key quicksort (MKQS) [4], itself a version of ternary-split quicksort (TSQS) [3], designed for sorting strings (any strings, not necessarily suffixes). Given an array of strings A, in k-order, MKQS sorts A as TSQS would, only it uses the symbols at position k as sort keys. MKQS chooses a pivot element, $p[k]$ for string $p \in A$, and splits A into three partitions $A_<$, $A_=$ and $A_>$. In $A_<$, the left partition, are placed the strings y with $y[k] < p[k]$; $A_>$, the right parition is filled with strings y with $y[k] > p[k]$; and $A_=$, the middle partition contains strings y with $y[k]$ equal to the pivot element. MKQS is then called recursively on the three partitions: $A_<$ and $A_>$ again use the symbols at position k as sort keys, but $A_=$ uses the symbols at $k+1$.

Practical implementations of MKQS (including ours) use insertion sort for small arrays (< 64 elements) to improve average case performance. Our implementation also diverts to heapsort when the recursion depth reaches a predefined level (48) — this technique is borrowed from [16] and helps with worst case performance. As the pivot element we choose the median of three or nine values depending on the number of strings in A.

Figure 1 gives high level pseudocode for our MKQS. We apply it to each 2-group of suffix pointers in SP in turn, according to the lexorder of the associated bigrams (ie. we sort group *aa* before *ab* and then *ac* and so on). In the next two sections we describe two powerful techniques that we layer on top of this primary sorting procedure to speed sorting when the strings are known to be suffixes of the same string, as it is with those in array SP.

3.4 Dynamic Sort Keys

The performance of MKQS can be improved specifically for sorting suffixes by modifying the values in ISA' (hence the sort keys) in a particular way as sorting proceeds.

For each bigram $b = x[i]x[i+1]$ we maintain a counter next_rank$[b]$ initially set to lo_rank$[b]$. When the final position of a sample suffix i prefixed with b becomes known, relative to the other sample suffixes in SP, ISA'$[i]$ is set to next_rank$[b]$ and next_rank$[b]$ is incremented. Because of the initial transform and the maintenance of next_rank counters, the MKQS continues to sort suffixes correctly in the face of the changing ISA' values. The

[4] We believe these requirements can be quite easily reduced to exactly $5n$ plus stack, but we do not investigate this claim further here.

```
MKQS(A,d)
  if |A| < t1 then
    insertion_sort(A,d)
  elsif d > t2 then
    heap_sort(A,d)
  else
    choose a partitioning element p
    partition A around p[d] to form
    arrays A<,A=,A>
    MKQS(A<,d)
    MKQS(A=,d + 1)
    MKQS(A>,d)
```

Figure 1: An overview of our Multi-key quicksort routine.

only other requirement for this to work correctly is that when suffix SP[i] is placed in its final position, suffixes in SP[$0..i-1$] are already sorted — this happens naturally anyway as a consequence of MKQS always recursing on the $A_<$ partition before $A_=$ and $A_>$.

Modifying ISA' on-the-fly in this way will ultimately lead to faster sorting as the number of unique symbols (sort keys) in ISA' increases. The idea shares something with the *induction sorting* technique described in [12], and the *order caching* trick of Seward [20]. The major difference here is that we store the sorted rank information so it is immediately available when it is needed and does not have to be retrieved from another part of memory — the MKQS procedure carries on regardless, and the use of rank information is "seemlessly" integrated. This means we incur far fewer cache misses than the methods of [12, 20].

3.5 Handling Repetitions

A repetition (tandem repeat) u^r in a string x is a substring $x[i..j]$ of length $|u|r = j - i + 1$ that consists of $r > 1$ concatenations of the same string u. We say the repetition has *period* $p = |u|$ and *exponent* r and that string u is the *generator* of the repetition. Suffixes that are prefixed with repetitions can be difficult to sort. For instance, suffix i prefixed with u^r and suffix $i + |u|$ prefixed with u^{r-1} must be compared to depth $|u^{r-1}|$ (at least) before their relative order is known, and when several suffixes are so prefixed the required effort is even greater. Fortunately it is possible to detect repetitions during sorting and, having detected them, sort them efficiently without the need for extraneous symbol comparisons. Here we provide the intuition for how repetitions are detected and processed, leaving out some nuances of the implementation.

Consider a subarray V of the SP array, V[$0..v$] = SP[$s..s + v$], which corresponds to the middle partition of the last quicksort round. The suffixes in V are in k-order, and so are k-equal. Let $u[0..k-1]$ be the length k common prefix of the suffixes in V.

Observation 3. *For each suffix $i \in V$, if $i + k$ is also in V, then suffix i is prefixed with a repetition u^r, with $u = x[i..i + k - 1]$.*

For large arrays it requires too much effort ($O(v^2)$) to check this condition for each suffix in V, so we instead test the following weaker condition.

Observation 4. *For each suffix $i \in V$ if $x[i] = x[i+k]$ then suffix i* might *be prefixed with a repetition.*

This condition can be checked in $O(v)$ time with a scan of V. Let b be the bigram that prefixes all the suffixes in V and let $\xi = \text{next\_rank}[b] + |V_=| - 1$. Note that ξ does not yet appear in ISA' due to the nature of the transform applied to the input. If the scan indicates V might contain suffixes prefixed with repetitions (by Observation 4) then for each $i \in V$ we set ISA'$[i] = \xi$. This effectively marks the start of every occurence of the substring $u[0..k-1] = \text{ISA}'[i..i+k]$. Now, increasing the sort depth to $k+1$, we split V into three partitions — $V_<$, $V_=$, $V_>$ — using ξ as the pivot element.

Lemma 5. *The suffixes in $V_=$ are prefixed with repetitions.*

Proof. Each suffix i in $V_=$ has ξ at position k which means $x[i+k..i+2k-1] = u$. Also, because i was in V, suffix i must have prefix $u = x[i..i+k-1]$, and so it has prefix u^2. □

The lexorder of the suffixes in $V_=$ can be determined efficiently from the lexorder of $V_<$ and $V_>$. Let substring $x[i..j] = u^r$ be a repetition: we call position $j - |u| - 1$ the *terminating position* of the repetition.

Lemma 6. *Consider suffixes $x[i..n] = u^{r_1}\alpha$ and $x[j..n] = u^{r_2}\beta$ with integers $r_1, r_2 > 2$ and non-empty strings α, β, u. Without loss of generality assume $r_1 < r_2$ and lexicographically $\alpha < u < \beta$. Lexicographically suffixes $i + |u|r_1 < i + |u|(r_1 - 1) < ... < i < j < j + |u| < ... < j + |u|r_2$.*

In words, the above lemma says that for a set of suffixes all prefixed with repetitions having the same generator, the order of the suffixes can be determined from the order of the terminating positions of the repetitions. The lexorder of $V_<$ and $V_>$ provides the lexorder of all the terminating positions of repetitions that are prefixes of suffixes in $V_=$. We sort $V_<$ and $V_>$ in turn and then scan left to right over the entire V array, enforcing Lemma 6: For each suffix i encountered in the scan if ISA'$[i-k] = \xi$ then place $i-k$ at the front of the $V_=$ partition and increment the front of the partition. Once placed suffix $i-k$ is in its final sorted position and can be assigned a unique value from next_rank[] as described in §3.4.

A nice feature of this heuristic is that it integrates so easily with the MKQS procedure. Also, suffixes prefixed with more than one repetition (with different period) are handled without special treatment in the recursion. We have found this technique very effective for suffix sorting in a genomic setting, where strings frequently contain many repetitions.

3.6 Finishing Off

Once the suffixes in SP are sorted, obtaining the complete SA is relatively easy. From Observation 2 it is easy to convert a transformed symbol/sort key ISA'$[i]$ into an original symbol $x[i]$ by using mod and bit-shift operations. In this way we convert the contents of ISA' at the end of sample sorting into the original string symbols.

We build the final SA over the top of ISA'. We want to move the contents of SP into a contiguous portion of ISA' and move x into SP. Recall each orginal symbol requires 1 byte each. We pack four symbols into each of the first $n/4$ integers of ISA', move the sample S (which requires no more than $2n$ bytes) from SP to ISA'$[n - |S|..n]$ and then relocate $x = \text{ISA}'[0..n/4]$ to SP. Now, using the ubiquitous bigram frequencies we can put each sample suffix into its final position in SA. All that remains is to scan SA and move non-sample suffixes into their final place, as described in §3.1.

4 Experimental Results

We raced an implementation of our algorithm, called `seemless`, against other leading suffix sorters. These were:

`ds` An implementation of the *deep-shallow* algorithm by [13] downloaded from `http://www.mfn.unipmn.it/~manzini/lightweight/`.

`bpr` An implementation of the *bucket-pointer-refinement* algorithm by [19] obtained from `http://bibiserv.techfak.uni-bielefeld.de/bpr/`.

`split` An implementation of the algorithm described in [12] available at `http://www.michael-maniscalco.com/msufsort.htm`.

`divsufsort` An implementation of the *improved two-stage* algorithm by [15] available at `http://homepage3.nifty.com/wpage/software/libdivsufsort.html`.

All programs were written in C or C++. We are confident that all implementations tested are of high quality. It should be noted that `split` is capable of sorting alphabets with $|\Sigma| > 256$ — we used an implementation tuned for sorting 8-bit alphabets. To assess practical performance we measured runtimes and memory usage of the implementations. The files used for testing were from the corpus compiled by Manzini[5] and Ferragina [13] listed in Table 2, which have become a defacto benchmark for testing suffix sorting algorithms.

All experiments were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. All running times given are the average of four runs and do not include time spent reading input files. Times were recorded with the standard Unix `time` function. Memory usage was recorded with the `memusage` command available with most Linux distributions.

Results are shown in Table 3. Our `seemless` algorithm is a clear winner on eight of the ten test files, going down narrowly to `divsufsort` on *jdk13c* and *reuters*. Our algorithm does best on file *w3c2*, where it is more than 20% faster than `divsufsort`. Figure 2 visualizes the resource requirements for each algorithm averaged over the entire test set. It shows that on average `seemless` runs faster than `divsufsort` at the cost of a slight increase in memory requirements. These two algorithms are roughly 20% faster than `split` and almost twice as fast as `ds`. In Figure 3 we report the ratios between the running time of `seemless` over `ds` and `divsufsort`. These ratios represent the time saved by `seemless` over the other two algorithms for each test file. For some inputs `seemless` is nearly 70% faster than `ds` (eg. on file *jdk13c*) and beats it by more than 20% on all but one of the files (*chr22*). These results indicate that `seemless` and `divsufsort` represent the state-of-the-art in lightweight suffix sorting for strings on byte-sized alphabets.

5 Concluding Remarks

In this paper we have introduced a new lightweight suffix array construction algorithm that, through a variety of techniques, performs very well in practice. Sampling means that for most strings only a small fraction (at most half, but often around a quarter) of all suffixes need

[5] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`

Table 2: Description of the data set used in experiments for small alphabet suffix array construction and character based BWT scenarios. LCP refers to the Longest Common Prefix amongst all suffixes in the string.

String	Mean LCP	Max LCP	Size (bytes)	Σ	Description
sprot34	89	7,373	109,617,186	66	SwissProt database
rfc	93	3,445	116,421,901	120	Concatenated IETF RFC files
howto	267	70,720	39,422,105	197	Linux Howto files
reuters	282	26,597	114,711,151	93	Reuters news in XML format
linux	479	136,035	116,254,720	256	Linux kernel source
jdk13c	679	37,334	69,728,899	113	JDK 1.3 documentation
etext99	1,108	286,352	105,277,340	146	Texts from Gutenberg project
chr22	1,979	199,999	34,553,758	4	Human chromosome 22
gcc	8,603	856,970	86,630,400	121	Gnu C Complier source
w3c2	42,300	990,053	104,201,579	255	HTML files from W3C site

Table 3: Runtime in milliseconds (in parentheses peak memory usage in bytes per input symbol) for the Suffix Array construction.

String	`seemless`	`split`	`ds`	`bpr`	`divsufsort`
sprot34	**48500** (5.18)	58990 (5.31)	74090 (5.01)	98980 (9.01)	50960 (5.00)
rfc	**47530** (5.04)	61310 (5.22)	65000 (5.01)	93190 (9.06)	48110 (5.00)
howto	**13810** (5.15)	18410 (5.25)	18160 (5.01)	21520 (9.78)	15170 (5.00)
reuters	56610 (5.22)	72920 (5.16)	146740 (5.01)	132930 (9.02)	**54220** (5.00)
linux	**40630** (5.08)	56620 (5.15)	55110 (5.01)	69570 (9.58)	41750 (5.00)
jdk13c	29880 (5.12)	43570 (5.14)	81880 (5.01)	63720 (9.08)	**28160** (5.00)
etext99	**49840** (5.25)	58160 (5.17)	75790 (5.01)	91760 (9.12)	52910 (5.00)
chr22	**14990** (5.08)	18100 (5.51)	15510 (5.01)	15970 (9.00)	15350 (5.00)
gcc	**29690** (5.08)	62120 (5.20)	73970 (5.01)	59810 (9.16)	31150 (5.00)
w3c2	**40640** (5.21)	65420 (5.17)	118370 (5.01)	84040 (9.64)	51260 (5.00)

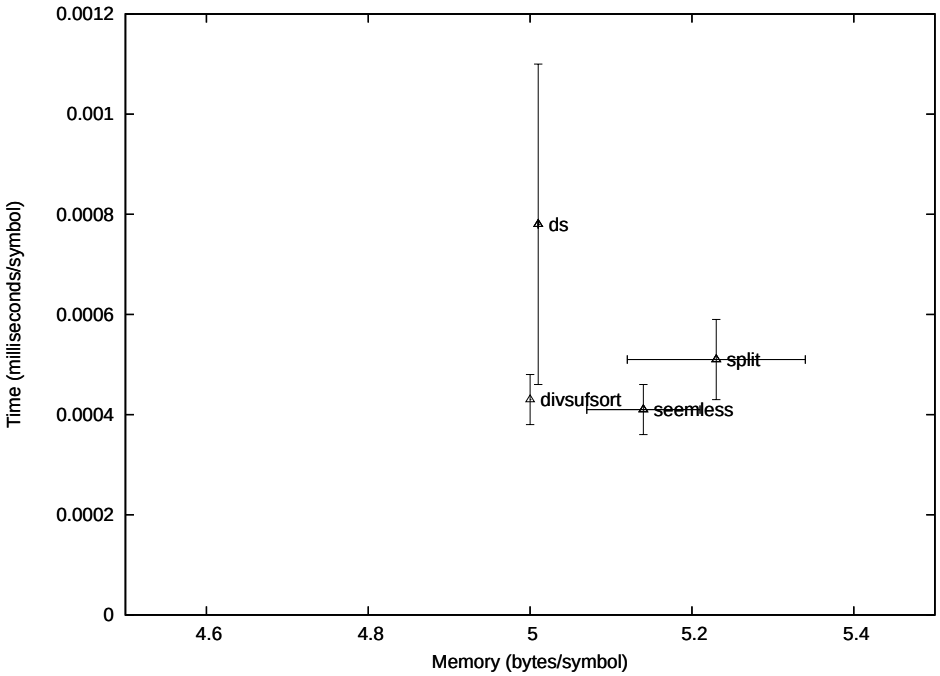

Figure 2: Resource requirements of the five algorithms when computing the suffix array on the Manzini corpus. Error bars are one standard deviation. Abscissa errors bars for `ds` and `divsufsort` are not shown as they are insignificantly small. The data point for `bpr` was not plotted to improve presentation — it lies out of view at (9.25,0.00078).

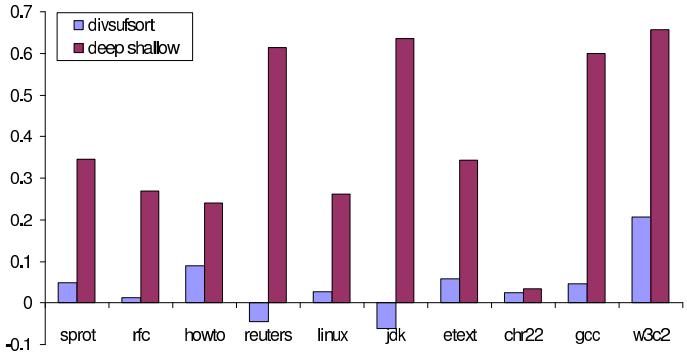

Figure 3: Running time reduction acheived by the `seemless` algorithm over `divsufsort` and `ds`. Each bar represents $1 - r$, where r is the ratio between the running time of `seemless` over the running time of `divsufsort` or `ds`.

to be sorted directly — the order of the rest can be induced from the sample. Placing ranks of sorted suffixes back in the string seemlessly speeds sorting of other suffixes and means our version of induction sorting incurs far less cache-misses than previous methods. The induction sorting idea itself improves cache-performance by quickly breaking the sample set down into smaller, more manageable arrays of k-ordered suffixes. Our repetitions heuristic deals with long runs of single characters and suffixes with high average lcp very effectively. In concert these three features also ensure the recursion stack stays very small on average.

We have steered clear of asymptotics in this paper but believe our repetitions heuristic could have a provable effect on asymptotics. Future work will focus on finding a theoretical bound on runtime. In this paper we have restricted Σ to having at most 256 symbols. Perhaps further restricting alphabet size would allow for even faster sorting, which might be of relevance in a genomic setting, where alphabets contain between four and 20 symbols. The same question goes for string length. We also wonder whether our algorithm can be adapted to sort an evenly sampled subset of suffixes, as is required for many compressed full text indices [10]. Another interesting question is whether the approach can be engineered to use $< 5n$ bytes of memory if we allow the SA's final destination to be on secondary storage. A C++ code implementation of the algorithm described in this paper is available from the authors.

References

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

[2] Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI, pages 85–96. Springer-Verlag, Berlin, 1985.

[3] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[4] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, 1997.

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

[6] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval*, pages 81–88, Cancun, Mexico, 1999. IEEE Computer Society.

[7] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloq. Automata, Languages and Programming*, volume 2971 of *Lecture Notes in Computer Science*, pages 943–955. Springer-Verlag, Berlin, 2003.

[8] D. K. Kim, Sim J. S., H. Park, and K. Park. Linear-time construction of suffix arrays. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, Berlin, 2003.

[9] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, Berlin, 2003.

[10] V. Mäkinen and G. Navarro. Compressed full text indexes. Technical Report TR/DCC-2005-7, Department of Computer Science, University of Chile, June 2006.

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

[12] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics*, 2006. to appear.

[13] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[14] E. M. McCreight. A space-economical suffix tree construction algroithm. *Journal of the ACM*, 23(2):262–272, 1976.

[15] Yuta Mori. DivSufSort, 2005. http://www.homepage3.nifty.com/wpage/software/ libdivsufsort.html.

[16] David R. Musser. Introspective sorting and selection algorithms. *Software - Practice and Experience*, 27(8):983–993, 1997.

[17] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. The performance of linear time suffix sorting algorithms. In M. Cohn and J. Storer, editors, *Proceedings of the IEEE Data Compression Conference*, pages 358–368, Los Alamitos, CA, March 2005. IEEE Computer Society Press.

[18] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. In *Proceedings of the Prague Stringology Conference*, pages 1–30, Prague, August 2005. Czech Technical University.

[19] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. In *Proceedings of The Seventh Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 77–85. SIAM, January 2005.

[20] J. Seward. On the performance of BWT sorting algroithms. In *Proceedings of the IEEE Data Compression Conference*, pages 173–182, Los Alamitos, CA, March 2000. IEEE Computer Society.

[21] E. Ukkonen. Online construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[22] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th annual Symposium on Foundations of Computer Science*, pages 1–11, 1973.