Engineering a Lightweight Suffix Array Construction Algorithm (Extended Abstract)*

Giovanni Manzini[1,2] and Paolo Ferragina[3]

[1] Dipartimento di Informatica, Università del Piemonte Orientale
I-15100 Alessandria, Italy
manzini@mfn.unipmn.it
[2] Istituto di Informatica e Telematica, CNR
I-56100 Pisa, Italy
[3] Dipartimento di Informatica, Università di Pisa
I-56100 Pisa, Italy
ferragin@di.unipi.it

1 Introduction

We consider the problem of computing the *suffix array* of a text $T[1, n]$. This problem consists in sorting the suffixes of T in lexicographic order. The suffix array [16] (or PAT array [9]) is a simple, easy to code, and elegant data structure used for several fundamental string matching problems involving both linguistic texts and biological data [4, 11]. Recently, the interest in this data structure has been revitalized by its use as a building block for three novel applications: **(1)** the Burrows-Wheeler compression algorithm [3], which is a provably [17] and practically [20] effective compression tool; **(2)** the construction of succinct [10, 19] and compressed [7, 8] indexes; the latter can store both the input text and its full-text index using roughly the same space used by traditional compressors for the text alone; and **(3)** algorithms for clustering and ranking the answers to user queries in web-search engines [22]. In all these applications the construction of the suffix array is the computational bottleneck both in time and space. This motivated our interest in designing *yet another* suffix array construction algorithm which is fast and "lightweight" in the sense that it uses small space.

The suffix array consists of n integers in the range $[1, n]$. This means that in theory it uses $\Theta(n \log n)$ bits of storage. However, in most applications the size of the text is smaller than 2^{32} and it is customary to store each integer in a four byte word; this yields a total space occupancy of $4n$ bytes. For what concerns the cost of constructing the suffix array, the theoretically best algorithms run in $\Theta(n)$ time [5]. These algorithms work by first building the suffix tree and then obtaining the sorted suffixes via an in-order traversal of the tree. However, suffix tree construction algorithms are both complex and space consuming since they occupy at least $15n$ bytes of working space (or even more, depending on the

* Partially supported by Italian MIUR project on "Technologies and services for enhanced content delivery".

R. Möhring and R. Raman (Eds.): ESA 2002, LNCS 2461, pp. 698–710, 2002.
© Springer-Verlag Berlin Heidelberg 2002

text structure [14]). This makes their use impractical even for moderately large texts. For this reason, suffix arrays are usually built using algorithms which run in $O(n \log n)$ time but have a smaller space occupancy. Among these algorithms the current "leader" is the qsufsort algorithm by Larsson and Sadakane [15]. qsufsort uses $8n$ bytes[1] and despite the $O(n \log n)$ worst case bound it is faster than the algorithms based on suffix tree construction.

Unfortunately, the size of our documents has grown much more quickly than the main memory of our computers. Thus, it is desirable to build a suffix array using as small space as possible. Recently, Itoh and Tanaka [12] and Seward [21] have proposed two new algorithms which only use $5n$ bytes. From the theoretical point of view these algorithms have a $\Theta(n^2 \log n)$ worst case complexity. In practice they are faster than qsufsort when the average LCP is small (the LCP is the length of the longest common prefix between two consecutive suffixes in the suffix array). However, for texts with a large average LCP these algorithms can be slower than qsufsort by a factor 100 or more.[2]

In this paper we describe and extensively test a new lightweight suffix sorting algorithm. Our main idea is to use a very small amount of extra memory, in addition to $5n$ bytes, to avoid the degradation in performance when the average LCP is large. To achieve this goal we make use of engineered algorithms and *ad hoc* data structures. Our algorithm uses $5n + cn$ bytes, where c is a user tunable parameter (in our tests c was at most 0.03). For files with average LCP smaller than 100 our algorithm is faster than Seward's algorithm and roughly two times faster than qsufsort. The best algorithm in our tests uses $5.03n$ bytes and is faster than qsufsort for all files except for the one with the largest average LCP.

2 Definitions and Previous Results

Let $T[1, n]$ denote a text over the alphabet Σ. The suffix array [16] (or PAT array [9]) for T is an array $SA[1, n]$ such that $T[SA[1], n]$, $T[SA[2], n]$, *etc.* is the list of suffixes of T sorted in lexicographic order. For example, for T = babcc then $SA = [2, 1, 3, 5, 4]$ since $T[2, 5]$ = abcc is the suffix with lower lexicographic rank, followed by $T[1, 5]$ = babcc, followed by $T[3, 5]$ = bcc and so on.[3]

Given two strings v, w we write LCP(v, w) to denote the length of their longest common prefix. The average LCP of a text T is defined as the average length of the longest common prefix between two consecutive suffixes. The average LCP is a rough measure of the difficulty of sorting the suffixes: if the average LCP is large we need in principle to examine "many" characters in order to establish the relative order of two suffixes.

[1] Here and in the following the space occupancy figures include the space for the input text, for the suffix array, and for any auxiliary data structure used by the algorithm.

[2] This figure refers to Seward algorithm [21]. We are in the process of acquiring the code of the Itoh-Tanaka algorithm and we hope we will be able to test it in the final version of the paper.

[3] Note that to define the lexicographic order of the suffixes it is customary to append at the end of T a special end-of-text symbol which is smaller than any symbol in Σ.

Table 1. Files used in our experiments sorted in order of increasing average LCP

Name	*Ave.* LCP	*Max.* LCP	*File size*	*Description*
bible	13.97	551	4,047,392	The file bible of the Canterbury corpus
e.coli	17.38	2,815	4,638,690	The file E.coli of the Canterbury corpus
wrld	23.01	559	2,473,400	The file world192 of the Canterbury corpus
sprot	89.08	7,373	109,617,186	Swiss prot database (sprot34.dat)
rfc	93.02	3,445	116,421,901	Concatenation of RFC text files
howto	267.56	70,720	39,422,105	Concatenation of Linux Howto text files
reuters	282.07	26,597	114,711,151	Reuters news in XML format
linux	479.00	136,035	116,254,720	Linux kernel 2.4.5 source files (tar archive)
jdk13	678.94	37,334	69,728,899	html and java files from the JDK 1.3 doc.
chr22	1,979.25	199,999	34,553,758	Assembly of human chromosome 22
gcc	8,603.21	856,970	86,630,400	gcc 3.0 source files (tar archive)

Since this is an "algorithmic engineering" paper we make the following assumptions which correspond to the situation most often faced in practice. We assume $|\Sigma| \leq 256$ and that each alphabet symbol is stored in one byte. Hence, the text $T[1,n]$ takes precisely n bytes. Furthermore, we assume that $n \leq 2^{32}$ and that the starting position of each suffix is stored in a four byte word. Hence, the suffix array $SA[1,n]$ takes precisely $4n$ bytes. In the following we use the term "lightweight" to denote a suffix sorting algorithm which use $5n$ bytes plus some small amount of extra memory (we are intentionally giving an informal definition). Note that $5n$ bytes are just enough to store the input text T and the suffix array SA. Although we do not claim that $5n$ bytes are indeed required, we do not know of any algorithm using less space.

For testing the suffix array construction algorithms we use the collection of files shown in Table 1. These files contain different kind of data in different formats; they also display a wide range of sizes and of average LCP's.

2.1 The Larsson-Sadakane qsufsort Algorithm

The qsufsort algorithm [15] is based on the doubling technique introduced in [13] and first used for the construction of the suffix array in [16]. Given two strings v, w and $t > 0$ we write $v <_t w$ if the length-t prefix of v is lexicographically smaller than the length-t prefix of w. Similarly we define the symbols $\leq_t, =_t$ and so on. Let s_1, s_2 denote two suffixes and assume $s_1 =_t s_2$ (that is, $T[s_1, n]$ and $T[s_2, n]$ have a length-t common prefix). Let $\hat{s}_1 = s_1 + t$ denote the suffix $T[s_1 + t, n]$ and similarly let $\hat{s}_2 = s_2 + t$. The fundamental observation of the doubling technique is that

$$s_1 \leq_{2t} s_2 \iff \hat{s}_1 \leq_t \hat{s}_2. \tag{1}$$

In other words, we can derive the $\leq_{2t}$ order between s_1 and s_2 by looking at the rank of $\hat{s}_1$ and $\hat{s}_2$ in the $<_t$ order.

The algorithm qsufsort works in rounds. At the beginning of the ith round the suffixes are already sorted according to the $\leq_{2^i}$ ordering. In the ith round the algorithm looks for groups of suffixes sharing the first 2^i characters and sorts them according to the $\leq_{2^i}$ ordering using Bentley-McIlroy ternary quicksort [1]. Because of (1) each comparison in the quicksort algorithm takes $O(1)$ time. After at most $\log n$ rounds all the suffixes are sorted. Thanks to a very clever data organization qsufsort only uses $8n$ bytes. Even more surprisingly, the whole algorithm fits in two pages of clean and elegant C code.

The experiments reported in [15] show that qsufsort outperforms other suffix sorting algorithm based on either the doubling technique or the suffix tree construction. The only algorithm which runs faster than qsufsort, but only for files with average LCP less than 20, is the Bentley-Sedgewick multikey quicksort [2]. Multikey quicksort is a *direct comparison* algorithm since it considers the suffixes as ordinary strings and sorts them via a character-by-character comparison without taking advantage of their special structure.

2.2 The Itoh-Tanaka two-stage Algorithm

In [12] Itoh and Tanaka describe a suffix sorting algorithm called two-stage suffix sort (two-stage from now on). two-stage only uses the text T and the suffix array SA for a total space occupancy of $5n$ bytes. To describe how it works, let us assume $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Using counting sort, two-stage initially partitions the suffixes into $|\Sigma|$ *buckets* $\mathsf{B}_\mathtt{a}, \ldots, \mathsf{B}_\mathtt{z}$ according to their first character. Note that a bucket is nothing more than a set of consecutive entries in the array SA which now is sorted according to the $\leq_1$ ordering. Within each bucket two-stage distinguishes between two types of suffixes: Type A suffixes in which the second character of the suffix is smaller than the first, and Type B suffixes in which the second character is larger than or equal to the first suffix character. The crucial observation of algorithm two-stage is that when all Type B suffixes are sorted we can derive the ordering of Type A suffixes. This is done with a single pass over the array SA; when we meet suffix $T[i, n]$ we look at suffix $T[i-1, n]$: if it is a Type A suffix we move it to the first empty position of bucket $\mathsf{B}_{T[i-1]}$.

Type B suffixes are sorted using textbook string sorting algorithms: in their implementation the authors use MSD radix sort [18] for sorting large groups of suffixes, Bentley-Sedgewick multikey quicksort for medium size groups, and insertion sort for small groups. Summing up, two-stage can be considered an "advanced" direct comparison algorithm since Type B suffixes are sorted by direct comparison whereas Type A suffixes are sorted by a much faster procedure which takes advantage of the special structure of the suffixes.

In [12] the authors compare two-stage with three direct comparison algorithms (quicksort, multikey quicksort, and MSD radix sort) and with an earlier version of qsufsort. two-stage turns out to be roughly 4 times faster than quicksort and MSD radix sort, and 2 to 3 times faster than multikey quicksort and qsufsort. However, the files used for the experiments have an average LCP of at

most 31, and we know that the advantage of doubling algorithms (like qsufsort) become apparent for much larger average LCP's.

2.3 Seward copy Algorithm

Independently of Itoh and Tanaka, in [21] Seward describes a lightweight algorithm, called copy, which is based on a concept similar to the Type A/Type B suffixes used by algorithm two-stage.

Using counting sort, copy initially sorts the array SA according to the $\leq_2$ ordering. As before we use the term *bucket* to denote the contiguous portion of SA containing a set of suffixes sharing the same first character. Similarly, we use the term *small bucket* to denote the contiguous portion of SA containing suffixes sharing the first two characters. Hence, there are $|\Sigma|$ buckets each one consisting of $|\Sigma|$ small buckets. Note that one or more (small) buckets can be empty.

copy sorts the buckets one at a time starting with the one containing the fewest suffixes, and proceeding up to the largest one. Assume for simplicity that $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. To sort a bucket, let us say bucket $\mathsf{B_p}$, copy sorts the small buckets $\mathsf{b_{pa}}, \mathsf{b_{pb}}, \ldots, \mathsf{b_{pz}}$. The crucial point of algorithm copy is that when bucket $\mathsf{B_p}$ is completely sorted, with a simple pass over it copy sorts all the small buckets $\mathsf{b_{ap}}, \mathsf{b_{bp}}, \ldots, \mathsf{b_{zp}}$. These small buckets are marked as sorted and therefore copy will skip them when their "parent" bucket is sorted. As a further improvement, Seward shows that even the sorting of the small bucket $\mathsf{b_{pp}}$ can be avoided since its ordering can be derived from the ordering of the small buckets $\mathsf{b_{pa}}, \ldots, \mathsf{b_{po}}$ and $\mathsf{b_{pq}}, \ldots, \mathsf{b_{pz}}$. This trick is extremely effective when working on files containing long runs of identical characters.

Algorithm copy sorts the small buckets using Bentley-McIlroy ternary quicksort. During this sorting the suffixes are considered atomic, that is, each comparison consists of the complete comparison of two suffixes. The standard trick of sorting the larger side of the partition last and eliminating tail recursion ensures that the amount of space required by the recursion stack grows, in the worst case, logarithmically with the size of the input text. In [21] Seward compares a tuned implementation of copy with the qsufsort algorithm on a set of files with average LCP up to 400. In these tests copy outperforms qsufsort for all files but one. However, Seward reports that copy is much slower than qsufsort when the average LCP exceeds a thousand, and for this reason he suggests the use of qsufsort as a fallback when the average LCP is large.[4]

Since the source code of both qsufsort and copy is available[5], we have tested both algorithms on our suite of test files which have an average LCP ranging from 13.97 to 8603.21 (see Table 1). The results of our experiments are reported in the top two rows of Table 2 (for a AMD Athlon processor) and Table 3 (for a Pentium III processor). In accordance with Seward's results, copy is faster than

[4] In [21] Seward describes another algorithm, called cache, which is faster than copy for files with larger average LCP. However, algorithm cache uses $6n$ bytes.

[5] Algorithm copy was originally conceived to split the input file into 1MB blocks. We modified it to allow the computation of the suffix array for the whole file.

qsufsort when the average LCP is small, and it is slower when the average LCP is large. The turning point appears to be when the average LCP is in the range 100-250. However, this is not the complete story. For example for all files the running time of qsufsort on the Pentium is smaller than the running time for the Athlon; this is not true for copy (see for example files *jdk13* and *gcc*). We conjecture that a difference in the cache architecture and behavior could explain this difference, and we plan to investigate it in the full paper (see also Section 4). We can also see that the difference in performance between the two algorithms does not depend on the average LCP alone. The DNA file *chr22* has a very large average LCP, nevertheless the two algorithms have similar running times. The file *linux* has a much greater average LCP than *reuters* and roughly the same size. Nevertheless, the difference in the running times between qsufsort and copy is smaller for *linux* than for *reuters*.

The most striking data in Tables 2 and 3 are the running times for *gcc*: for this file algorithm copy is 150-200 times slower than qsufsort. This is not acceptable since *gcc* is not a pathological file built to show the weakness of copy, on the contrary it is a file downloaded from a very busy site and we can expect that there are other files like it on our computers.[6] In the next section we describe a new algorithm which uses several techniques for avoiding such catastrophic behavior and at the same time retaining the nice features of copy: the $5n$ bytes space occupancy and the good performance for files with moderate average LCP.

3 Our Contribution: Deep-Shallow Suffix Sorting

Our starting point for the design of an efficient suffix array construction algorithm is Seward copy algorithm. Within this algorithm we replace the procedure used for sorting the small buckets (i.e. the groups of suffixes having the first two characters in common). Instead of using Bentley-McIlroy ternary quicksort we use a more sophisticated technique. More precisely, we sort the small buckets using Bentley-Sedgewick multikey quicksort and we stop the recursion when we reach a predefined depth L (that is, when we have to sort a group of suffixes with a length-L common prefix). At this point we switch to a different string sorting algorithm. This approach has several advantages: **(1)** it provides a simple and efficient mean to detect the groups of suffixes with a long common prefix; **(2)** because of the limit L, the size of the recursion stack is bounded by a predefined constant which is independent of the size of the input text and can be tuned by the user; **(3)** if the suffixes in the small bucket have common prefixes which never exceed L, all the sorting is done by multikey quicksort which is an extremely efficient string sorting algorithm.

We call this approach to suffix sorting *deep-shallow* sorting since we mix an algorithm for sorting suffixes with small LCP (*shallow sorter*) with an algorithm (actually more than one, as we shall see) for sorting suffixes with large LCP (*deep*

[6] As we have already pointed out, algorithm copy was conceived to work on blocks of data of size at most 1MB. The reader should be aware that we are using an algorithm outside its intended domain!

Table 2. Running times (in seconds) for a 1400 MHz AMD Athlon processor, with 1GB main memory and 256Kb L2 cache. The operating system was Debian GNU/Linux Debian 2.2. The compiler was gcc ver. 2.95.2 with options -O3 -fomit-frame-pointer. The table reports (user + system) time averaged over five runs. The running times do not include the time spent for reading the input files

	bible	*e.coli*	*wrld*	*sprot*	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*chr22*	*gcc*
qsufsort	5.44	5.21	3.27	313.41	321.15	80.23	391.08	262.26	218.14	55.08	199.63
copy	3.36	4.55	1.69	228.47	201.06	68.25	489.75	297.26	450.93	48.69	28916.45
ds0 $L=500$	2.64	3.21	1.29	157.54	139.71	50.36	294.57	185.79	227.94	33.38	2504.98
ds0 $L=1000$	2.57	3.22	1.29	157.04	140.11	50.26	292.25	185.00	235.74	33.27	2507.15
ds0 $L=2000$	2.66	3.23	1.29	157.00	139.93	50.30	292.35	185.46	237.75	33.29	2511.50
ds0 $L=5000$	2.66	3.23	1.31	156.90	139.87	50.36	291.47	185.53	239.48	33.23	2538.78
ds1 $L=200$	2.51	3.21	1.29	169.68	149.12	41.76	301.10	150.35	148.14	33.44	343.02
ds1 $L=500$	2.51	3.22	1.28	161.94	147.35	40.62	309.97	140.85	177.28	33.32	295.70
ds1 $L=1000$	2.51	3.22	1.29	157.60	145.12	40.52	298.23	138.11	202.28	33.27	289.40
ds1 $L=2000$	2.50	3.19	1.27	157.19	140.93	41.10	291.18	139.06	202.30	33.18	308.41
ds1 $L=5000$	2.51	3.18	1.28	157.09	139.73	42.76	289.95	145.74	212.77	33.21	372.35
ds2 $d=500$	2.64	3.19	1.35	157.09	139.34	37.48	292.22	121.15	164.97	33.33	230.64
ds2 $d=1000$	2.55	3.19	1.28	157.33	139.14	38.50	284.50	124.07	184.86	33.30	242.99
ds2 $d=2000$	2.50	3.18	1.27	156.93	139.81	39.67	286.56	128.26	191.71	33.25	266.27
ds2 $d=5000$	2.51	3.19	1.28	157.05	139.65	41.94	289.78	137.08	210.01	33.31	332.55

sorter). In the next sections we describe several deep sorting strategies, i.e. algorithms for sorting suffixes which have a length-L common prefix.

3.1 Blind Sorting

Let $s_1, s_2, \ldots, s_m$ denote a group of m suffixes with a length-L common prefix that we need to deep-sort. If m is small (we will discuss later what this means) we sort them using an algorithm, called *blind sort*, which is based on the *blind trie* data structure introduced in [6, Sect. 2.1] (see Fig. 1). Blind sorting simply consists in inserting the strings $s_1, \ldots, s_m$ one at a time in an initially empty blind trie; then we traverse the trie from left to right thus obtaining the strings sorted in lexicographic order.

The insertion of string s_i in the trie requires a first phase in which we scan s_i and simultaneously traverse the trie until we reach a leaf ℓ. Then we compare s_i with the string associated to leaf ℓ and we determine the length of their common prefix. Finally, we update the trie adding the leaf corresponding to s_i (see [6] for details and for the merits of blind tries with respect to standard compacted tries). Obviously in the construction of the trie we ignore the first L characters of each suffix because they are identical.

Our implementation of the blind sort algorithm uses at most $36m$ bytes of memory. Therefore, we use it when the number of suffixes to be sorted is less

Table 3. Running times (in seconds) for a 1000MHz Pentium III processor, with 1GB main memory and 256Kb L2 cache. The operating system was GNU/Linux Red Hat 7.1. The compiler was gcc ver. 2.96 with options -O3 -fomit-frame-pointer. The table reports (user + system) time averaged over five runs. The running times do not include the time spent for reading the input files

	bible	*e.coli*	*wrld*	*sprot*	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*chr22*	*gcc*
qsufsort	4.96	4.63	3.16	230.52	245.02	64.71	290.20	213.81	168.59	42.69	162.68
copy	3.34	4.63	1.76	230.10	197.73	70.82	532.02	324.29	519.10	47.33	35258.04
ds0 $L=500$	2.28	2.96	1.24	122.84	114.50	47.60	246.87	192.02	218.85	26.02	3022.47
ds0 $L=1000$	2.21	2.80	1.20	122.29	114.10	47.62	243.98	191.32	221.65	26.07	3035.91
ds0 $L=2000$	2.21	2.80	1.19	121.80	113.80	48.51	242.71	192.33	222.68	26.04	3026.48
ds0 $L=5000$	2.28	2.94	1.23	121.55	113.72	47.80	242.17	186.40	225.59	25.99	3071.17
ds1 $L=200$	2.29	2.99	1.26	137.27	124.75	36.07	253.79	140.10	127.52	26.28	383.03
ds1 $L=500$	2.18	2.85	1.20	127.29	122.13	34.49	262.66	126.61	150.68	26.10	331.50
ds1 $L=1000$	2.20	2.85	1.20	122.76	119.27	34.60	248.58	123.38	174.54	26.06	325.91
ds1 $L=2000$	2.18	2.79	1.19	121.50	114.85	35.23	240.59	124.24	175.30	25.93	344.11
ds1 $L=5000$	2.19	2.80	1.20	121.80	113.53	37.39	240.42	132.50	190.77	26.05	410.72
ds2 $d=500$	2.18	2.79	1.20	121.66	112.74	30.95	239.44	102.51	133.37	25.95	249.23
ds2 $d=1000$	2.18	2.79	1.19	121.44	112.57	32.16	232.45	105.79	152.49	25.92	262.48
ds2 $d=2000$	2.22	2.81	1.20	121.65	113.35	33.99	235.94	111.35	162.75	26.05	287.02
ds2 $d=5000$	2.23	2.82	1.20	121.54	113.25	36.86	239.99	121.88	186.84	25.99	353.70

than $B = \frac{n}{2000}$. Thus, the space overhead of using blind sort is at most $\frac{9n}{500}$ bytes. If the text is 100MB long, this overhead is 1.8MB which should be compared with the 500MB required by the text and the suffix array.[7]

If the number of suffixes to be sorted is larger than $B = \frac{n}{2000}$ we sort them using Bentley-McIlroy ternary quicksort. However, with respect to algorithm copy, we introduce the following two improvements:

1. As soon as we are working with a group of suffixes smaller than B we stop the recursion and we sort them using blind sort;
2. during each partitioning phase we compute L_S (resp. L_L) which is the longest common prefix between the pivot and the strings which are lexicographically smaller (resp. larger) than the pivot. When we sort the strings which are smaller (resp. larger) than the pivot we can skip the first L_S (resp. L_L) characters since we know they constitute a common prefix.

We have called the above algorithm ds0 and its performances are reported in Tables 2 and 3 for several values of the parameter L (the depth at which we

[7] Although we believe this is a small overhead, we point out that the limit $B = \frac{n}{2000}$ was chosen somewhat arbitrarily. Preliminary experimental results show that there is only a marginal degradation in performance when we take $B = \frac{n}{3000}$, or $B = \frac{n}{4000}$. In the future we plan to better investigate the space/time tradeoff introduced by this parameter and its impact on the cache performance.

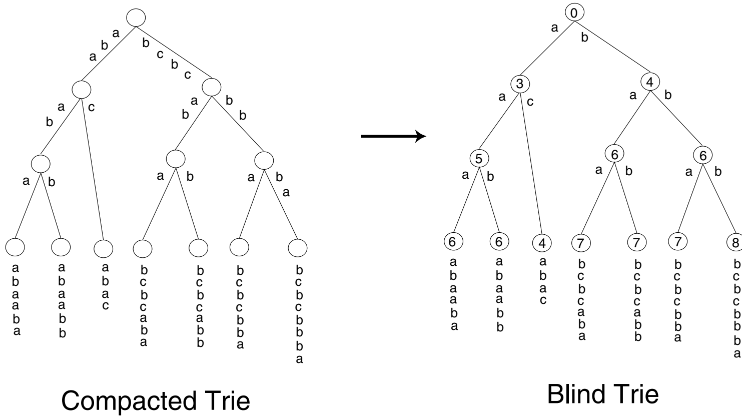

Fig. 1. A standard compacted trie (left) and the corresponding blind trie (right) for the strings: abaaba, abaabb, abac, bcbcaba, bcbcabb, bcbcbba, bcbcbbba. Each internal node of the blind trie contains an integer and a set of outgoing labelled arcs. A node containing the integer k represent a set of strings which have a length-k common prefix and differ in the $(k+1)$st character. The outgoing arcs are labelled with the different characters that we find in position $k+1$. Note that since the outgoing arcs are ordered alphabetically, by visiting the trie leaves from left to right we get the strings in lexicographic order

stop multikey quicksort and we switch to the blind sort/quicksort algorithms). We can see that algorithm ds0 is slower than qsufsort only for the files *jdk13* and *gcc*. If we compare copy and ds0 we notice that our deep-shallow approach has reduced the running time for *gcc* by a factor 10. This is certainly a good start. As we shall see, we will be able to reduce it again by the same factor taking advantage of the fact that the strings we are sorting are all suffixes of the same text.

3.2 Induced Sorting

One of the nice features of two-stage and copy algorithms is that some of the suffixes are not sorted by direct comparison: instead their relative order is derived in constant time from the ordering of other suffixes which have been already sorted. We use a generalization of this technique in the deep-sorting phase of our algorithm. Assume we need to sort the suffixes $s_1, \ldots, s_m$ which have a length-L common prefix. We scan the first L characters of s_1 looking at each pair of consecutive characters (e.g. $T[s_1]T[s_1+1]$, $T[s_1+1]T[s_1+2]$, up to $T[s_1+L-2]T[s_1+L-1]$). As soon as we find a pair of characters, say $\alpha\beta$, belonging to an already sorted small bucket $\mathsf{b}_{\alpha\beta}$ the ordering of $s_1, \ldots, s_m$ can be derived from the ordering of $\mathsf{b}_{\alpha\beta}$ as follows.

Assume $\alpha = T[s_1+t]$ and $\beta = T[s_1+t+1]$ for some $t < L-1$. Since $s_1, \ldots, s_m$ have a length-L common prefix, every s_i contains the pair $\alpha\beta$ starting from position t. Hence $\mathsf{b}_{\alpha\beta}$ contains m suffixes corresponding to $s_1, \ldots, s_m$ (that is, $\mathsf{b}_{\alpha\beta}$ contains the suffixes starting at $s_1 + t, s_2 + t, \ldots, s_m + t$). Note that these suffixes are not necessarily consecutive in $\mathsf{b}_{\alpha\beta}$. Since the first $t - 1$ characters of $s_1, \ldots, s_m$ are identical, the ordering of $s_1, \ldots, s_m$ can be derived from the ordering of the corresponding suffixes in $\mathsf{b}_{\alpha\beta}$. Summing up, the ordering is done as follows:

1. We sort the suffixes $s_1, \ldots, s_m$ according to their starting position in the input text $T[1, n]$. This is done so that in Step 3 we can use binary search to answer membership queries in the set $s_1, \ldots, s_m$.
2. Let $\hat{s}$ denote the suffix starting at the text position $T[s_1 + t]$. We scan the small bucket $\mathsf{b}_{\alpha\beta}$ in order to find the position of $\hat{s}$ within $\mathsf{b}_{\alpha\beta}$.
3. We scan the suffixes preceding and following $\hat{s}$ in the small bucket $\mathsf{b}_{\alpha\beta}$. For each suffix s we check whether the suffix starting at the position $T[s - t]$ is in the set $s_1, \ldots, s_m$; if so we mark the suffix s.[8]
4. When m suffixes in $\mathsf{b}_{\alpha\beta}$ have been marked, we scan them from left to right. Since $\mathsf{b}_{\alpha\beta}$ is sorted this gives us the correct ordering of $s_1, \ldots, s_m$.

Obviously there is no guarantee that in the length-L common prefix of $s_1, \ldots,$ s_m there is a pair of characters belonging to an already sorted small bucket. In this case we simply resort to the quicksort/blind sort combination. We call this algorithm ds1 and its performances are reported in Tables 2 and 3 for several values of L. We can see that ds1 with $L = 500$ runs faster than qsufsort for all files except *gcc*. In general, ds1 appears to be slightly slower than ds0 for files with small average LCP but it is clearly faster for the files with large average LCP: for *gcc* it is 8-9 times faster.

3.3 Anchor Sorting

Profiling shows that the most costly operation of induced sorting is the scanning of the small bucket $\mathsf{b}_{\alpha\beta}$ in search of the position of suffix $\hat{s}$ (Step 2 above). We now show that we can avoid this operation if we are willing to use a small amount of extra memory. For a fixed $d > 0$ we partition the text $T[1, n]$ into n/d segments of length d: $T[1, d], T[d+1, 2d]$ and so on up to $T[n-d+1, n]$ (for simplicity let us assume that d divides n). We define two arrays Anchor$[\cdot]$ and Offset$[\cdot]$ of size n/d such that, for $i = 1, \ldots, n/d$:

- Offset$[i]$ contains the position of leftmost suffix which starts in the ith segment and belongs to an already sorted small bucket. If in the ith segment does not start any suffix belonging to an already sorted small bucket then Offset$[i] = 0$.
- Let $\hat{s}_i$ denote the suffix whose starting position is stored Offset$[i]$. Anchor$[i]$ contains the position of $\hat{s}_i$ within its small bucket.

[8] The marking is done setting the most significant bit of s. This means that we can work with texts of size at most 2^{31}. The same restriction holds for qsufsort as well.

The use of the arrays Anchor[·] and Offset[·] is fairly simple. Assume that we need to sort the suffixes $s_1, \ldots, s_m$ which have a length-L common prefix. For $j = 1, \ldots, m$, let t_j denote the segment containing the starting position of s_j. If $\hat{s}_{t_j}$ (that is, the leftmost already sorted suffix in segment t_j) starts within the first L characters of s_j (that is, $s_j < \hat{s}_{t_j} < s_j + L$) then we can sort $s_1, \ldots, s_m$ using the induced sorting algorithm described in the previous section. However, we can skip Step 2 since the position of $\hat{s}_{t_j}$ within its small bucket is stored in Anchor[t_j]. Obviously, it is possible that for some j $\hat{s}_{t_j}$ does not exist or cannot be used. However, since the suffixes $s_1, \ldots, s_m$ usually belong to different segments, we have m possible candidates. In our implementation among the available sorted suffixes $\hat{s}_{t_j}$'s we use the one whose starting position is closest to the corresponding s_j (that is, we choose j which minimizes $\hat{s}_{t_j} - s_j$; this helps Step 3 of induced sorting). If there is no available sorted suffix, then we resort to the blind sort/quicksort combination.

For what concerns the space occupancy of anchor sorting, we note that in Offset[i] we can store the distance between the beginning of the ith segment and the leftmost sorted suffix. Hence Offset[i] $< d$. If we take $d < 2^{16}$ the array Offset requires $2n/d$ bytes of storage. Since each entry of Anchor requires four bytes, the overall space occupancy is $6n/d$ bytes. In our tests we used at least $d = 500$ which yields an overhead of $\frac{6n}{500}$ bytes. If we add the $\frac{9n}{500}$ bytes required by blind sorting with $B = \frac{n}{2000}$, we get a maximum overhead of at most $\frac{3n}{100}$ bytes. Hence, for a 100MB text the overhead is at most 3MB, which we consider a "small" amount compared with the 500MB used by the text and the suffix array.

In Tables 2 and 3 we report the running times of anchor sorting (under the name ds2) for d ranging from 500 to 5000 and $L = d + 50$. We see that for the files with moderate average LCP ds2 with $d = 500$ is significantly faster than copy and roughly two times faster than qsufsort. For the files with large average LCP ds2 is faster than qsufsort for all files except *gcc*. For *gcc* ds2 is 15% slower than qsufsort on the Athlon and 50% slower on the Pentium. In our opinion this slowdown on a single file is an acceptable price to pay in exchange for the reduction in space occupancy achieved over qsufsort ($5.03n$ bytes vs. $8n$ bytes). We believe that the possibility of building suffix arrays for larger files has more value than a greater efficiency in handling files with a very large average LCP.

4 Conclusions and Further Work

In this paper we have presented a novel algorithm for building the suffix array of a text $T[1, n]$. Our algorithm uses $5.03n$ bytes and is faster than any other tested algorithm. Only on a single file our algorithm is outperformed by qsufsort which however uses $8n$ bytes.

For pathological inputs, i.e. texts with an average LCP of $\Theta(n)$, all lightweight algorithms take $\Theta\left(n^2 \log n\right)$ time. Although this worst case behavior does not occur in practice, it is an interesting theoretical open question whether we can achieve $O(n \log n)$ time using $o(n)$ space in addition to the space required by the input text and the suffix array.

References

[1] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software – Practice and Experience*, 23(11):1249–1265, 1993. 701

[2] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997. 701

[3] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. 698

[4] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994. 698

[5] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. 698

[6] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999. 704

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000. 698

[8] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001. 698

[9] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates and, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992. 698, 699

[10] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the 32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000. 698

[11] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. 698

[12] 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, SPIRE '99*, pages 81–88. IEEE Computer Society Press, 1999. 699, 701

[13] R. Karp, R. Miller, and A. Rosenberg. Rapid Identification of Repeated Patterns in Strings, Arrays and Trees. In *Proceedings of the ACM Symposium on Theory of Computation*, pages 125–136, 1972. 700

[14] S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999. 699

[15] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999. 699, 700, 701

[16] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. 698, 699, 700

[17] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. 698

[18] P. M. McIlroy and K. Bostic. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993. 701

[19] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceeding of the 11th International Symposium on Algorithms and Computation*, pages 410–421. Springer-Verlag, LNCS n. 1969, 2000. 698
[20] J. Seward. The BZIP2 home page, 1997. `http://sourceware.cygnus.com/bzip2/index.html`. 698
[21] J. Seward. On the performance of BWT sorting algorithms. In *DCC: Data Compression Conference*, pages 173–182. IEEE Computer Society TCC, 2000. 699, 702
[22] O. Zamir and O. Etzioni. Grouper: A dynamic clustering interface to web search results. *Computer Networks*, 31(11-16):1361–1374, 1999. 698