Technical Report TR-INF-2003-02-02-UNIPMN

Engineering a lightweight suffix array construction algorithm*

Giovanni Manzini† — Università del Piemonte Orientale

Paolo Ferragina‡ — Università di Pisa

Abstract

In this paper we describe a new algorithm for building the suffix array of a string. This task is equivalent to the problem of lexicographically sorting all the suffixes of the input string. Our algorithm is based on a new approach called deep-shallow sorting: we use a "shallow" sorter for the suffixes with a short common prefix, and a "deep" sorter for the suffixes with a long common prefix.

All the known algorithms for building the suffix array either require a large amount of space or are inefficient when the input string contains many repeated substrings. Our algorithm has been designed to overcome this dichotomy. Our algorithm is "lightweight" in the sense that it uses very small space in addition to the space required by the suffix array itself. At the same time our algorithm is fast even when the input contains many repetitions: this has been shown by extensive experiments with inputs of size up to 110MB.

The source code of our algorithm, as well as a C library providing a simple API, is available under the Gnu GPL [21].

1 Introduction

In this paper we consider the problem of computing the *suffix array* of a text string $T[1,n]$. This problem consists in sorting the suffixes of T in lexicographic order. The suffix array [19] (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, 12]. Recently, the interest in this data structure has been revitalized by its use as a building block for two novel applications: **(1)** the Burrows-Wheeler compression algorithm [3], which is a provably [20] and practically [24] effective compression tool; and **(2)** the construction of succinct [11, 23] or compressed [7, 8, 10] indexes. In these applications the construction of the suffix array is the computational bottleneck both

*Partially supported by the Italian MIUR projects "Algorithmics for Internet and the Web (ALINWEB)" and "Technologies and Services for Enhanced Content Delivery". A preliminary version of this work has appeared on the Proc. of the 10th European Symposium on Algorithms (ESA '02).

†Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italy and IIT-CNR, Pisa, Italy. E-mail: `manzini@mfn.unipmn.it`.

‡Dipartimento di Informatica, Università di Pisa, Italy. E-mail: `ferragina@di.unipi.it`.

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 working space.

The suffix array consists of n integers in the range $[1, n]$. This means that in principle 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 known 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 text structure [17]). This makes their use impractical even for moderately large texts. For this reason, suffix arrays are usually built directly 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 [18]. qsufsort uses $8n$ bytes[1] and—despite the $O(n \log n)$ worst case bound—it is much faster in practice 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 [13] and Seward [25] have proposed two new algorithms which only use $5n$ bytes. We call these algorithms *lightweight algorithms* to stress their (relatively) small space occupancy. 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 (Longest Common Prefix) is small. However, for texts with a large average LCP these algorithms can be slower than qsufsort by a factor 100 or more.

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 any 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 can be chosen by the user at run time; in our tests c was at most 0.03. The theoretical worst case complexity of our algorithm is still $\Theta(n^2 \log n)$, but its behavior in practice is quite good. Extensive experiments, carried out on four different architectures, show that our algorithm is faster than any other tested algorithm. Only on a single instance—a single file on a single architecture—our algorithm was outperformed by qsufsort.

2 Definitions and previous results

Let $T[1, n]$ denote a text over the alphabet Σ. The suffix array [19] (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 the lowest lexicographic rank, followed by $T[1, 5] =$ babcc, followed by $T[3, 5] =$ bcc and so on.[2]

Given two strings v, w we write $\text{LCP}(v, w)$ 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

[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] 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.

Name	*Ave.* LCP	*Max.* LCP	*File size*	*Description*
sprot	89.08	7,373	109,617,186	Swiss prot database (original file name 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	Tar archive containing the Linux kernel 2.4.5 source files
jdk13	678.94	37,334	69,728,899	Concatenation of html and java files from the JDK 1.3 doc.
etext99	1,108.63	286,352	105,277,340	Concatenation of Project Gutemberg etext99/*.txt files
chr22	1,979.25	199,999	34,553,758	Genome assembly of human chromosome 22
gcc	8,603.21	856,970	86,630,400	Tar archive containing the gcc 3.0 source files
w3c	42,299.75	990,053	104,201,579	Concatenation of html files from www.w3c.org

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

two consecutive suffixes, that is

$$\text{Average LCP} = \left(\frac{1}{n-1}\right) \sum_{i=1}^{n-1} \text{LCP}(T[SA[i], n], T[SA[i+1], n]).$$

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.

In the rest of the 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 $T[1, n]$ occupies 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 $SA[1, n]$ occupies precisely $4n$ bytes. In the following we use the term "lightweight" to denote a suffix sorting algorithm which uses $5n$ 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 kinds 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 [18] is based on the doubling technique introduced in [15] and first used for the construction of the suffix array in [19]. 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 $\leq_t$ and $=_t$. 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

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

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 $\leq_t$ 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 at groups of suffixes sharing the first 2^i characters and sorts them according to the $\leq_{2^{i+1}}$ 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 [18] show that qsufsort outperforms other suffix sorting algorithms 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. In this paper we did not consider multikey quicksort since it is well known that it is inefficient when the average LCP is large. However, for inputs with a small average LCP it is one of the fastest algorithms: see [16] for an efficient suffix sorting algorithm based on multikey quicksort.

2.2 The Itoh-Tanaka two-stage algorithm

In [13] 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 $\Sigma = \{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\}$ and let SA be initialized as $SA[i] = i$. Using counting sort, two-stage initially sorts the array SA according to the $\leq_1$ ordering. Then it logically partitions SA into $|\Sigma|$ *buckets* $\mathsf{B}_\mathsf{a}, \ldots, \mathsf{B}_\mathsf{z}$. A bucket is a set of consecutive entries of SA containing the suffixes which start with the same character, from a to z in our illustrative example. 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. Within each bucket two-stage stores Type A suffixes first, followed by Type B suffixes. This is correct since Type A suffixes lexicographically precede Type B suffixes.

The crucial observation of algorithm two-stage is that when all Type B suffixes are sorted, we can easily derive the ordering of the Type A suffixes. This can be done with a single pass over the array SA: when we meet suffix $s_i = T[i, n]$ we look at suffix $s_{i-1} = T[i-1, n]$, if s_{i-1} 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 [22] 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 [13] 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 from 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) with respect to direct comparison algorithms becomes apparent for much larger average LCP's.

Some improvements to algorithm two-stage have been recently described in [14]. Although these improvements are based on some interesting algorithm ideas, we do not describe them here since they lead to an algorithm which is not lightweight—its space requirement being $9n$ bytes.

2.3 Seward copy algorithm

Independently of Itoh and Tanaka, Seward describes in [25] 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. We use the term *sub-bucket* to denote the contiguous portion of SA containing suffixes sharing the first two characters. There are $|\Sigma|$ buckets, each one consisting of $|\Sigma|$ sub-buckets. One or more (sub-)buckets can be empty. In the following we use the symbol B_α to denote the bucket containing the suffixes starting with character α, and we use the symbol $\mathsf{b}_{\alpha\beta}$ to denote the sub-bucket containing the suffixes starting with the character-pair $\alpha\beta$.

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 $\Sigma = \{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\}$. To sort a bucket, say B_p, copy sorts the sub-buckets $\mathsf{b}_\mathsf{pa}, \mathsf{b}_\mathsf{pb}, \ldots, \mathsf{b}_\mathsf{pz}$ individually. The crucial point of algorithm copy is that when bucket B_p is completely sorted, with a simple pass over it copy sorts all the sub-buckets $\mathsf{b}_\mathsf{ap}, \mathsf{b}_\mathsf{bp}, \ldots, \mathsf{b}_\mathsf{zp}$. These sub-buckets are marked as sorted and copy skips them when their "parent" bucket is sorted. In other words, assuming B_a is sorted after B_p, when we sort B_a we skip b_ap and any other already sorted sub-bucket within B_a.

As a further improvement, Seward shows that even the sorting of the sub-bucket b_pp can be avoided since its ordering can be derived from the ordering of the sub-buckets $\mathsf{b}_\mathsf{pa}, \ldots, \mathsf{b}_\mathsf{po}$ and $\mathsf{b}_\mathsf{pq}, \ldots, \mathsf{b}_\mathsf{pz}$. This trick, first suggested in [3], is extremely effective when working on files containing long runs of identical characters.

Algorithm copy sorts the sub-buckets using Bentley-McIlroy ternary quicksort. During this sorting the suffixes are considered atomic, that is, each comparison consists of the scanning of two entire suffixes. The standard trick of sorting the largest 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 [25] 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.

2.4 Seward cache algorithm

In [25] Seward describes how to improve algorithm copy in order to better deal with files with large average LCP. The new algorithm, called cache, uses an auxiliary array $R[1, n]$ of sixteen bit integers. Initially all entries in R are set to zero. When the sorting of a bucket B is completed, for each suffix $T[k, n]$ in B we write in $R[k]$ the most significant sixteen bits of its rank (the rank r of $T[k, n]$ is its position in the sorted suffix array, that is, $SA[r] = k$).

If at any point in the algorithm we are comparing the suffixes $T[i, n]$ and $T[j, n]$ we can proceed as follows. If $T[i] = T[j]$ we compare $R[i]$ and $R[j]$: if they differ we have the correct ordering of $T[i, n]$ and $T[j, n]$. If $R[i] = R[j]$, we next compare $T[i + 1]$ and $T[j + 1]$; if they are equal we can compare $R[i + 1]$ and $R[j + 1]$, and so on. Note that we use R to determine the ordering of suffixes which have the same first character. Hence, we can store in $R[i]$ the rank of $T[i,n]$ relative to its bucket. That is, we do not need the absolute rank of $T[i,n]$, but only its rank among the suffixes starting with the character $T[i]$.

In the experiments reported in [25], copy was faster than qsufsort and cache for files with small average LCP (up to 30). cache was the fastest algorithm for files with large average LCP, and qsufsort was the fastest only for a single file with an average LCP of 33.77. However, in the experiments of [25] the input files were split in blocks of size 10^6 bytes, and the maximum average LCP was 383.17; this explains the relatively poor performance of qsufsort which is efficient when the average LCP is large.

Algorithm cache as described above uses $7n$ bytes: $5n$ for T and SA plus $2n$ for R. Since we are interested in lightweight algorithms we have modified it in order to reduce its space occupancy to $6n$ bytes. This has been achieved by defining $R[1,n]$ as an array of eight bit integers. Clearly, this reduces the effectiveness of R: now we can only store the eight most significant bits of the ranks, and therefore ties are more likely when we compare the values stored in R. To compensate for this, we store in R ranks *relative to the sub-buckets*. Hence, as soon as the sub-bucket $\mathsf{b}_{T[k]T[k+1]}$ is sorted, we store in $R[k]$ the eight most significant bits of the rank of $T[k,n]$ within $\mathsf{b}_{T[k]T[k+1]}$. In the following we write cache_6n to denote this modified cache algorithm.

2.5 Preliminary experimental results

We have tested the three algorithms qsufsort, copy and cache_6n (our space economical version of cache) on our suite of test files (see Table 1). We have used two machines with different architectures: a 1000 MHz Pentium III with 256KB L2 cache, and a 933 MHz PowerPC G4 with 256KB L2 cache and 2MB L3 cache (the L3 cache runs at half the processor speed). The results of our experiments are reported in the top three rows of Table 2 for the Pentium and Table 3 for the PowerPC. The same data is represented as histograms in Fig. 1. Note that the test files are order by increasing average LCP.

For what concerns the relative performances of the three algorithms our results are in accordance with Seward's observations reported in [25]. copy is faster than qsufsort when the average LCP is small, and it is slower when the average LCP is large. cache_6n is faster than qsufsort roughly half of the times but there is no clear relationship between their relative speed and the average LCP of the input files.

If we compare the data in Tables 2 and 3 we see that all algorithms run faster on the Pentium than on the PowerPC with the exception of algorithm copy on the files with the largest average LCP. This is again in accordance with Seward's analysis of the algorithms qsufsort, copy, and cache. In [25, Sec. 5.3] Seward has shown that qsufsort does many random accesses to the memory and therefore does not fully benefit of the processor cache. This is true, to a lesser extent, also for cache; whereas copy was the algorithm generating the smallest number of cache misses. Thus, it is to be expected that copy benefits of the large L3 cache of the PowerPC; and indeed the phenomenon is more noticeable when the average LCP is large, since in this case most of the work of copy consists in comparing pairs of suffixes by means of sequential scans.

The data in Tables 2 and 3 also show that the hardness of building the suffix array does not depend on the average LCP and file size alone. For example, the file *reuters* has an average LCP smaller than *linux* and roughly the same size. Nevertheless, building the suffix array for *reuters* takes more time for all algorithms. On the PowerPC, building the suffix array for *reuters* with qsufsort and cache_6n takes more time than for the file *w3c* which has an average LCP 150 times

	sprot	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*etext99*	*chr22*	*gcc*	*w3c*
qsufsort	233.0	245.4	64.3	297.1	214.7	173.8	229.8	42.8	164.9	325.1
cache_6n	238.1	202.1	49.2	424.7	233.2	222.3	213.8	54.2	3533.5	271.7
copy	208.1	174.3	62.5	509.0	302.8	509.4	838.3	41.2	35577.4	23180.7
ds0 $L = 500$	121.6	113.8	47.3	245.5	189.1	217.4	571.5	25.9	3018.0	18137.0
ds0 $L = 1000$	121.1	113.4	47.4	242.8	191.6	221.3	571.4	26.1	3021.6	18107.7
ds0 $L = 2000$	121.1	113.0	47.0	241.4	192.8	221.8	571.3	25.8	3038.1	18290.6
ds0 $L = 5000$	121.0	112.7	47.5	241.1	190.6	226.1	578.9	25.8	3042.4	18024.2
ds1 $L = 500$	126.6	121.6	34.6	261.7	126.3	149.7	307.0	26.0	331.2	982.6
ds1 $L = 1000$	121.8	118.8	34.5	247.4	122.5	173.2	232.3	25.9	325.3	507.0
ds1 $L = 2000$	121.3	114.1	35.3	240.0	123.5	174.4	197.3	25.9	343.0	405.6
ds1 $L = 5000$	121.2	113.3	37.4	239.5	131.1	189.8	227.1	25.9	410.0	488.0
ds2 $d = 500$	121.1	112.2	30.9	238.7	101.6	131.7	126.8	25.9	248.8	269.0
ds2 $d = 1000$	121.1	111.7	32.0	231.9	105.3	152.2	139.7	25.9	261.5	228.9
ds2 $d = 2000$	121.2	112.9	33.6	234.7	110.3	162.1	162.3	25.9	286.9	270.1
ds2 $d = 5000$	120.9	113.0	36.5	238.7	120.6	186.5	212.2	25.9	356.4	409.1

Table 2: 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. The test files are ordered by increasing average LCP.

larger.

Another phenomenon worth mentioning is the behavior of copy and cache_6n on the files *gcc* and *w3c*. *w3c* is 20% larger than *gcc* and its average LCP is five times larger. Surprisingly, building the suffix array for *gcc* seems to be a more difficult task for copy and cache_6n; for cache_6n the running time for *gcc* is more than *ten times* larger than the time taken on *w3c* (note that in Fig. 1 the histograms for copy and cache_6n on *gcc* have been truncated). A few experiments have shown that the performances of copy and cache_6n on *gcc* can be improved using a better pivot selection strategy in Bentley-McIlroy ternary quicksort (which is used to sort the sub-buckets). However, we have not been able to fully disclose this apparently counterintuitive behavior.[3] Note that qsufsort shows the expected behavior: its running time for *gcc* is roughly half the running time for *w3c*.

Summing up, the data in Tables 2 and 3 show that qsufsort is a very fast and robust algorithm. Its only downside is that it uses $8n$ space. cache_6n—which only uses $6n$ space—is also quite fast, but its behavior on *gcc* suggests that it is not as robust as qsufsort. Finally, if we are tight on space and we are forced to use copy we must be prepared to wait a long time for files with a large average LCP: for *gcc* and *w3c* copy is 100-150 times slower than qsufsort.

In the next section we describe a new lightweight algorithm which retains the nice features of copy—small space occupancy and good performance for files with moderate average LCP—without suffering from a significant slowdown when the average LCP is large.

3 Our contribution: deep-shallow suffix sorting

Our starting point for the design of an efficient lightweight suffix array construction algorithm is Seward copy algorithm. Within this algorithm we replace the procedure used for sorting the sub-

[3] As we have already pointed out, algorithms copy and cache were conceived and engineered to work on blocks of data of size at most 1MB. They are not to be blamed if they are occasionally inefficient on inputs of size 80MB and more!

	sprot	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*etext99*	*chr22*	*gcc*	*w3c*
qsufsort	401.8	397.1	83.7	509.5	330.0	232.7	360.2	53.0	213.0	507.1
cache_6n	301.9	249.7	61.5	524.6	278.7	279.9	273.6	66.8	3393.6	322.8
copy	257.4	215.1	77.7	592.0	347.3	533.4	774.0	51.3	28288.7	18006.3
ds0 $L = 500$	170.6	152.0	56.6	343.0	207.3	233.4	491.7	38.1	1922.6	12587.5
ds0 $L = 1000$	170.2	151.9	56.3	339.4	206.3	242.1	495.3	38.1	1915.1	12569.4
ds0 $L = 2000$	170.0	151.3	56.2	337.6	206.2	243.1	498.9	38.1	1926.1	12580.5
ds0 $L = 5000$	169.9	151.1	56.1	336.9	206.0	245.1	511.5	38.1	1952.4	12565.1
ds1 $L = 500$	175.7	160.5	44.5	362.5	152.6	188.1	316.8	38.2	255.5	777.6
ds1 $L = 1000$	171.0	157.5	44.3	346.7	148.8	216.8	267.6	38.1	250.4	454.9
ds1 $L = 2000$	170.1	152.4	44.7	337.0	149.6	215.0	247.0	38.1	265.9	391.7
ds1 $L = 5000$	170.0	151.0	46.3	335.4	156.6	222.0	275.3	38.1	318.1	465.0
ds2 $d = 500$	170.3	151.4	41.0	341.4	131.2	176.2	187.5	40.3	210.2	301.4
ds2 $d = 1000$	179.8	150.7	42.0	329.5	131.0	194.3	195.1	38.1	212.5	250.0
ds2 $d = 2000$	170.0	151.2	43.2	331.7	135.6	200.9	215.1	38.1	230.7	284.8
ds2 $d = 5000$	170.0	151.0	45.5	334.6	144.6	218.3	260.3	38.1	284.4	396.3

Table 3: Running times (in seconds) for a 933MHz PowerPC G4 processor, with 1GB main memory, 256Kb L2 cache and 2MB L3 cache. The operating system was GNU/Linux Mandrake 8.2. The compiler was gcc ver. 2.95.3 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. The test files are ordered by increasing average LCP.

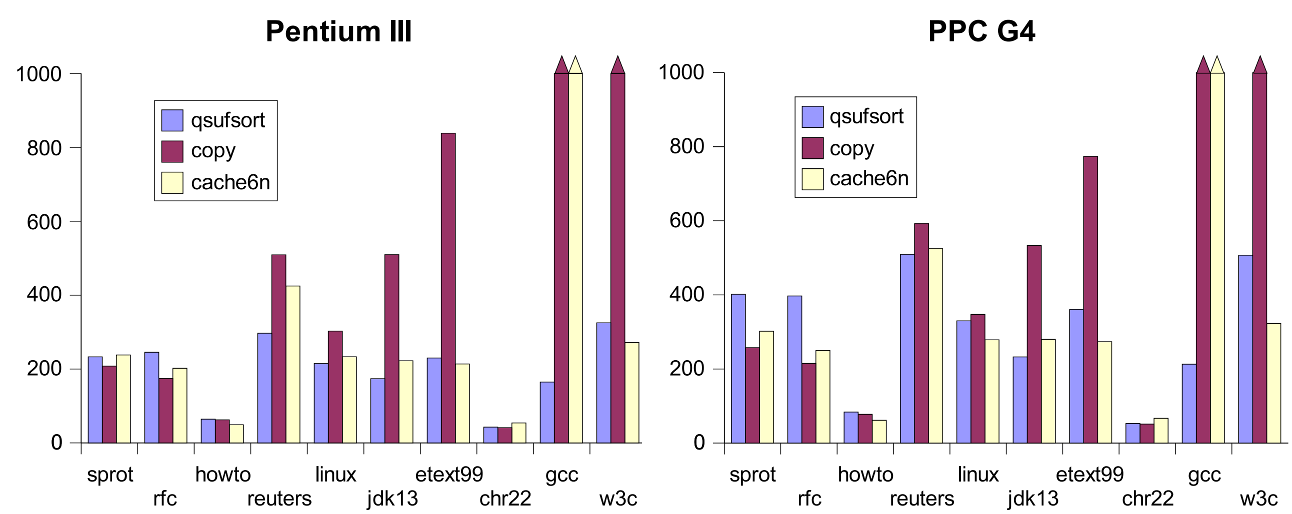

Figure 1: Graphical representations of the running times of qsufsort, copy, and cache_6n reported in Tables 2 and 3. Note that the histograms for copy on *gcc* and *w3c* and for cache_6n on *gcc* have been truncated since the running times are well beyond the upper limit of the Y axis. The test files are ordered by increasing average LCP.

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. We sort the sub-buckets using Bentley-Sedgewick multikey quicksort stopping 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 sub-bucket have common prefixes which never exceed L, their sorting is done by multikey quicksort which is an extremely efficient string sorting algorithm when the average LCP is small (see the last paragraph of Section 2.1).

We call this approach *deep-shallow* suffix sorting since we mix an algorithm for sorting suffixes with short LCP (*shallow sorter*) with an algorithm (actually more than one, as we shall see) for sorting suffixes with long LCP (*deep sorter*). In the next sections we describe several deep sorting strategies, that is, algorithms for sorting suffixes having a common prefix longer than L.

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. 2). 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. Obviously in the construction of the trie we ignore the first L characters of each suffix since we know that they are identical.

The insertion of string s_i in the trie consists of a first phase in which we scan s_i and simultaneously traverse top-down the trie until we reach a leaf ℓ. Then we compare s_i with the string, say s_j, associated to leaf ℓ and we determine the length of their common prefix. This length and the mismatching character allow to identify the position in the trie where the new leaf corresponding to s_i has to be inserted (see [6] for details). The crucial point of the algorithm is that for the insertion of s_i in the trie the only operations involving the suffixes $s_1, \ldots, s_i$ are:[4]

1. a *sequential access* to s_i during the traversal of the trie, and
2. the *sequential scan* of s_i and s_j during their comparison.

Thus, our algorithm sorts the suffixes using only "cache-friendly" sequential string scans. Note that we are neglecting in this analysis the cost of trie traversal because the trie is small, since m is chosen to be small, and thus the cost of suffix comparisons dominates the cost of trie percolation.

We point out that also the string based Bentley-McIlroy ternary quicksort algorithm, used within copy, sorts the suffixes by means of sequential scans. However, ternary quicksort executes on average $\Theta(m \log m)$ sequential scans, whereas our blind sorting algorithm executes only $\Theta(m)$ sequential

[4]In the following we use the expression "sequential access to s" when an algorithm reads the characters $s[j_1], s[j_2], \ldots, s[j_k]$ with $j_1 < j_2 < \cdots < j_k$. We use the expression "sequential scan" when an algorithm reads consecutive characters: $s[0], s[1], \ldots, s[k]$.

accesses to the suffixes. This improvement over ternary quicksort is payed in terms of the extra memory required for storing the trie data structure. This means that we cannot use blind sorting for an arbitrarily large group of suffixes.

Our implementation of blind sort uses at most $36m$ bytes of memory. We use it when the number of suffixes to be sorted is less than $B = \frac{n}{2000}$. With this choice 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.[5]

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 the ternary quicksort algorithm used within 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 call ds0 the suffix sorting algorithm which uses multikey quicksort up to depth L and then switches to the blind-sort/ternary-quicksort combination described above. The performance of ds0 are reported in Tables 2 and 3 for several values of the parameter L.

We can see that ds0 is faster than qsufsort and cache_6n on *chr22* and on the five files with the smallest average LCP. We can also see that ds0 is always faster than copy and that for the file *gcc* ds0 achieves a tenfold running time reduction. This is certainly a good start. We now show how to further reduce the running times by 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 the algorithms two-stage, copy, and cache_6n is that some of the suffixes are not sorted by direct comparison: 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, namely $T[s_1+i]T[s_1+i+1]$ for $i = 0, \ldots, L-1$. As soon as we find a pair of characters, say $\alpha\beta$, belonging to an already sorted sub-bucket $\mathsf{b}_{\alpha\beta}$, we derive the ordering of $s_1, \ldots, s_m$ from the ordering of $\mathsf{b}_{\alpha\beta}$ as follows.

Let $\alpha = T[s_1+t]$ 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 character-pair $\alpha\beta$ starting at 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 $s_1 + t, s_2 + t, \ldots, s_m + t$. The good news is that the first $t - 1$ characters of $s_1, \ldots, s_m$ are identical, so that the ordering of $s_1, \ldots, s_m$ can be derived from the ordering of the corresponding suffixes in $\mathsf{b}_{\alpha\beta}$. The bad news is that these corresponding suffixes are not necessarily consecutive in $\mathsf{b}_{\alpha\beta}$, even if they are expected to be close to each other because of their long common prefix. Combining these observations we derive the ordering of $s_1, \ldots, s_m$ as follows:

[5] Although we believe this is a small overhead, we point out that the limit $B = \frac{n}{2000}$ was chosen somewhat arbitrarily. Experimental results show that there is only a marginal degradation in performance when we take $B = \frac{n}{3000}$, or $B = \frac{n}{4000}$.

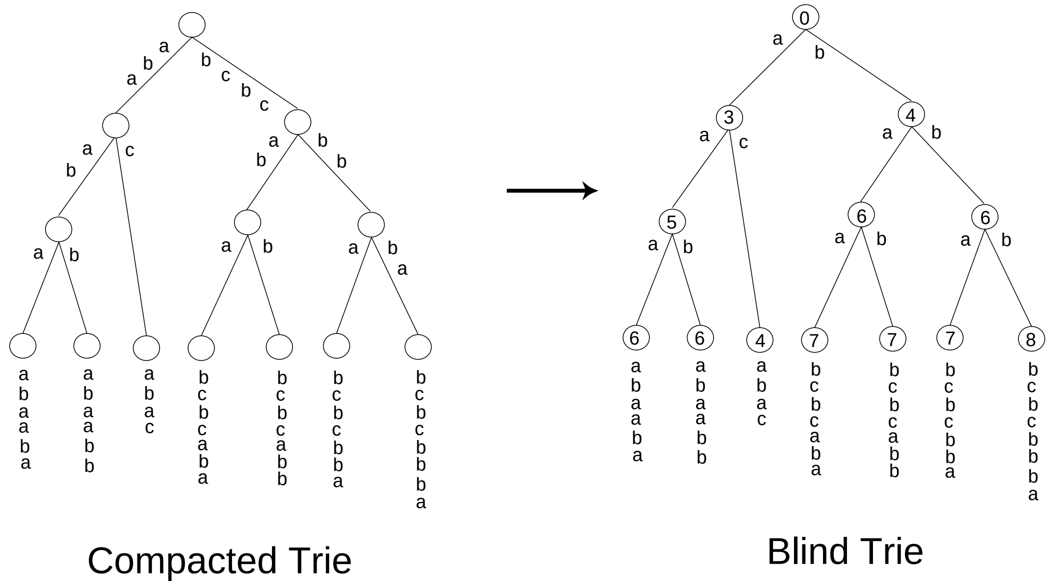

Figure 2: A standard compacted trie (left) and the corresponding blind trie (right) for the strings: abaaba, abaabb, abac, bcbcaba, bcbcabb, bcbcbba, bcbcbbba. Each internal node of the blind trie contains an integer and a set of outgoing labeled arcs. A node containing the integer k represent a set of strings which have a length-k common prefix and differ in the $(k+1)$st character. The outgoing arcs are labeled 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.

1-is 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-is below we can use binary search to answer membership queries in the set $s_1, \ldots, s_m$.

2-is Let $\hat{s}$ denote the suffix starting at the text character $T[s_1 + t]$. We scan the sub-bucket $\mathsf{b}_{\alpha\beta}$ in order to find the position of $\hat{s}$ within $\mathsf{b}_{\alpha\beta}$.

3-is We scan the suffixes preceding and following $\hat{s}$ in the sub-bucket $\mathsf{b}_{\alpha\beta}$. For each suffix s we check whether the suffix starting at the character $T[s - t]$ is in the set $s_1, \ldots, s_m$; if so we mark the suffix s.[6]

4-is 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 $s_1, \ldots, s_m$.

The effectiveness of the above procedure depends on how many suffixes are scanned at Step 3-is before all the suffixes corresponding to $s_1, \ldots, s_m$ are found and marked. We expect that this number is small since, as we already observed, the suffixes corresponding to $s_1, \ldots, s_m$ are expected to be close to each other in $\mathsf{b}_{\alpha\beta}$.

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 sub-bucket. In this case we cannot use induced sorting and we resort to the blind-sort/quicksort combination.

[6]We mark the suffixes by setting the most significant bit of the integer which represent the suffix s. This means that our algorithm can work with texts of size at most 2^{31} bytes. Note that the same restriction holds for qsufsort as well.

We call ds1 the algorithm which uses induced sorting and we report its performance for several values of L in Tables 2 and 3. 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 *w3c* it is more than ten times faster. We can see that ds1 with $L = 2000$ runs faster than qsufsort and cache_6n for all files except *gcc* and *w3c*.

3.3 Anchor sorting

Profiling shows that the most costly operation of induced sorting is the scanning of the sub-bucket $\mathsf{b}_{\alpha\beta}$ to search for the position of suffix $\hat{s}$ (Step 2-is above). We show how to avoid this operation using a small amount of extra memory. We partition the text $T[1, n]$ into n/d segments of length d: $T[1, d], T[d + 1, 2d]$ and so on (for simplicity we assume that d divides n). We define two arrays Anchor[·] and Offset[·] of size n/d such that:

- 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 in Offset[i]. Anchor[i] contains the position of $\hat{s}_i$ within its small bucket.

Note that the arrays Offset and Anchor provide a sort of partial inverse of the (already computed portion of the) suffix array. In this sense they are similar to the array $R[\cdot]$ used by cache and cache_6n which stores the most significant bits of the ranks of the already sorted suffixes.

The use of the arrays Anchor[·] and Offset[·] within induced sorting 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 z_j denote the segment containing the starting position of s_j. If $\hat{s}_{z_j}$ (that is, the leftmost already sorted suffix in segment z_j) starts within the first L characters of s_j (that is, $s_j < \hat{s}_{z_j} < s_j + L$) then we can sort $s_1, \ldots, s_m$ using the induced sorting algorithm described in the previous section. However, we can now skip Step 2-is since the position of $\hat{s}_{z_j}$ within its sub-bucket is stored in Anchor[z_j].

Obviously it is possible that, for some j, $\hat{s}_{z_j}$ does not exist or cannot be used because precedes s_j or follows $s_j + L$. 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 $\hat{s}_{z_j}$'s, we use the one whose starting position is closest to the corresponding s_j, that is, we choose j which minimizes $\hat{s}_{z_j} - s_j > 0$. This choice helps Step 3-is of induced sorting since—using the notation of Step 3-is—it minimizes the number of suffixes s such that the suffix starting at $T[s - t]$ *is not* in the set $s_1, \ldots, s_m$. If, for $j = 1, \ldots, m$, $\hat{s}_{z_j}$ does not exist or cannot be used then we resort to the blind-sort/quicksort combination.

For the updating of the arrays Offset and Anchor we use the following strategy. The straightforward approach is to update them each time we complete the sorting of a sub-bucket. Instead we update them at the end of *each call to deep sorting*, that is, each time we complete the sorting of a set of suffixes which share a length-L common prefix. This approach has a twofold advantage:

- Updates are done only when we have useful data. As an example, if a sub-bucket is sorted by shallow sorting alone, that is, all suffixes differ within the first L characters, the suffixes in that small bucket are not used to update Offset and Anchor. The rationale is that these suffixes are not very useful for induced sorting. It is easy to see that they can be used only for determining the ordering of suffixes which differ within the first $d + L$ characters while we

	sprot	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*etext99*	*chr22*	*gcc*	*w3c*
qsufsort	280.6	305.6	73.2	348.9	245.7	197.7	301.0	49.0	182.7	345.7
cache_6n	257.4	225.2	51.9	424.4	240.9	221.6	236.8	58.4	2601.7	269.2
ds2 $d = 500$	150.9	134.9	35.8	274.1	116.1	154.9	156.8	31.5	189.2	237.4
ds2 $d = 1000$	150.4	132.9	36.3	261.5	117.9	173.1	164.5	31.7	202.6	199.8
ds2 $d = 2000$	150.3	132.9	37.0	261.1	120.8	172.6	178.4	31.4	223.0	221.7
ds2 $d = 5000$	150.2	132.9	38.6	263.2	127.4	186.9	210.1	31.3	286.4	296.0

	sprot	*rfc*	*howto*	*reuters*	*linux*	*jdk13*	*etext99*	*chr22*	*gcc*	*w3c*
qsufsort	340.0	367.6	90.4	430.4	311.0	243.4	344.2	55.4	228.1	442.0
cache_6n	265.8	237.6	61.9	415.4	245.9	407.8	256.5	61.6	2171.1	464.3
ds2 $d = 500$	163.6	137.5	39.3	305.3	120.8	186.3	165.4	33.3	162.4	219.6
ds2 $d = 1000$	163.1	135.9	39.4	294.4	121.1	203.7	172.1	33.0	169.6	189.7
ds2 $d = 2000$	163.2	135.7	39.9	292.7	122.5	204.6	191.5	33.3	189.8	203.2
ds2 $d = 5000$	163.1	134.5	41.2	293.3	126.4	213.9	218.3	33.0	221.6	245.6

Table 4: Running times (in seconds) for a 1400 MHz Athlon XP (top) and a 1700 MHz Pentium 4 (bottom). Both machines were equipped with with 1GB main memory and 256Kb L2 cache. The operating system on the Athlon was GNU/Linux Debian 2.2; the compiler was gcc ver. 2.95.2 with options -O3 -fomit-frame-pointer. The operating system on the Pentium 4 was GNU/Linux Mandrake 9.0; the compiler was gcc ver. 3.2 with options -O3 -fomit-frame-pointer -march=pentium4. The table reports (user + system) time averaged over five runs. The running times do not include the time spent for reading the input files. The test files are ordered by increasing average LCP.

know that induced sorting is advantageous only when used for suffixes which have a very long common prefix.

- Updates are done as early as possible. When we complete the sorting of a set of suffixes $s_1, \ldots, s_m$ which share a length-L common prefix, we use them to update the arrays Offset and Anchor without waiting for the completion of the sorting of their sub-bucket. This means that anchor sorting can use $s_1, \ldots, s_m$ for determining the order of a set of suffixes which are in the same sub-bucket as $s_1, \ldots, s_m$.

For what concerns the space occupancy of anchor sorting, we observe that in Offset[i] we can store the distance between the beginning of the ith segment and the leftmost sorted suffix in the segment. Hence Offset[i] is always smaller than the segment length d. If we take $d < 2^{16}$ we can store the array Offset in $2n/d$ bytes. Since each entry of Anchor requires four bytes, the overall space occupancy is $6n/d$ bytes. In our tests d was at least 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 time of anchor sorting—under the name ds2—for d ranging from 500 to 5000 and $L = d + 50$. In Table 4 we report the running time of qsufsort, cache_6n, and ds2 on an Athlon XP and a Pentium 4. In Fig. 3 we show a graphical comparison of the running times of qsufsort, cache_6n, and ds2 with $d = 500$.

We can see that for the files with moderate average LCP ds2 with $d = 500$ is significantly faster than copy and cache_6n and roughly two times faster than qsufsort. For the files with large average LCP, ds2 is always faster than cache_6n and it is faster than qsufsort for all files except *gcc*. For *gcc* ds2 is faster than qsufsort on the Pentium 4 and slower on the Pentium III; on the PowerPC and the Athlon the two algorithms have roughly the same speed.

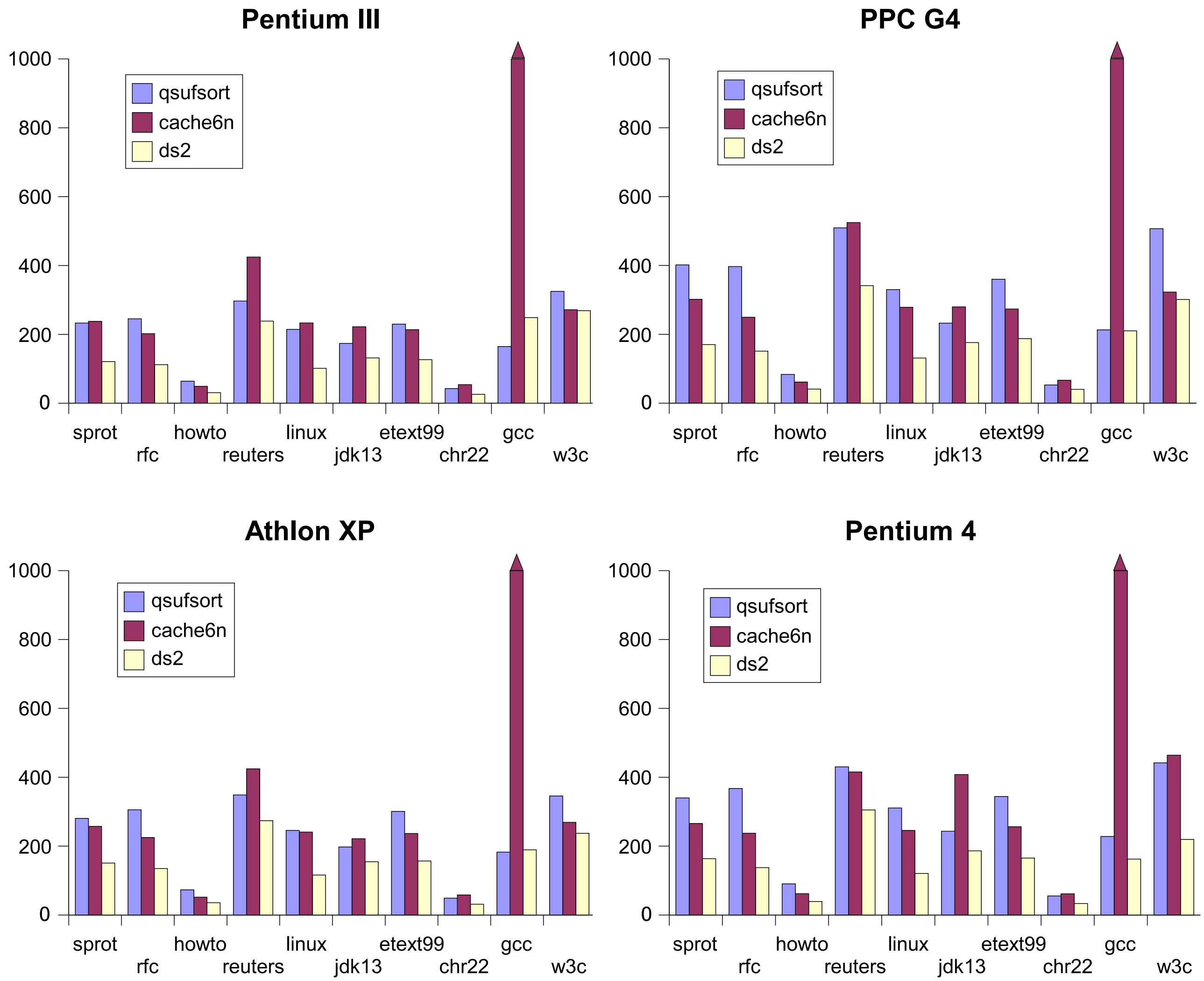

Figure 3: Graphical representations of the running times of qsufsort, cache_6n, and ds2 (with $d = 500$, $L = 550$) reported in Tables 2–4. Note that the histograms for cache_6n on *gcc* have been truncated since the running times are well beyond the upper limit of the Y axis. The test files are ordered by increasing average LCP.

A comment on the performance on the Pentium 4 is in order. We can see that most of the times the 1700MHz Pentium 4 is significantly slower than the 1000MHz Pentium III. Remarkable exceptions are cache_6n on *gcc* and ds2 on *gcc* and *w3c* for which the Pentium 4 is clearly faster. These data show once more that the architectures of modern CPU's can have significant and unexpected impacts on the execution speed of the different algorithms.[7]

In order to have a different perspective on the performances of ds2, in Fig. 4 we report the ratios between the running times of ds2 and qsufsort on the four different machines used in our tests. These ratios represent the reduction in running time achieved by ds2 over qsufsort. We observe that for all files except *gcc* the ratios for the Pentium III and the Athlon are quite close. We can also see

[7] Another peculiarity of the Pentium 4, is that the use of the compiler option -march=pentium4 greatly enhanced the performances of cache_6n and ds2 (it did not affect the performances of qsufsort). On the other machines, the -march option did not bring a clear improvement and therefore it was not used.

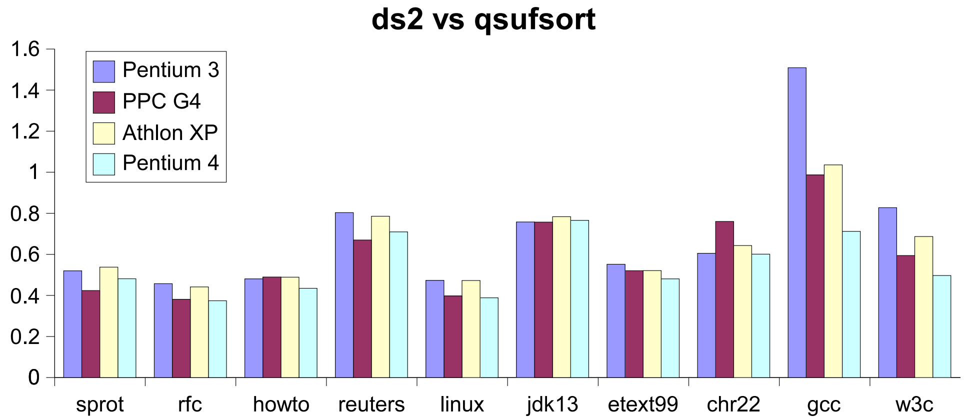

Figure 4: Running time reduction achieved by ds2 over qsufsort. Each bar represents the ratio between the running time of ds2 (with $d = 500$, $L = 550$) over the running time of qsufsort.

that—for all files except *chr22*—the smallest ratio are achieved on the PowerPC and the Pentium 4. This means that ds2 is "more efficient" than qsufsort on these architectures; however the difference is not marked and does not appear to be related with the average LCP of the input files.

Overall, the data reported in this section show the validity of our deep-shallow suffix sorting approach. We have been able to improve the already impressive performances of copy and cache_6n for files with moderate average LCP. At the same time we have avoided any significant degradation in performances for files with large average LCP: we are faster than any other algorithm with the only exception of the file *gcc* on the Pentium 3. We stress that this improvement in terms of running time has been achieved with a simultaneous reduction of the space occupancy. ds2 with $d = 500$ uses $5.03n$ space, cache_6n uses $6n$ space and qsufsort uses $8n$ space.

4 Concluding remarks

In this paper we have presented a lightweight algorithm for building the suffix array of a text $T[1, n]$. We have been motivated by the observation that the major drawback of most suffix array construction algorithms is their large space occupancy. Our algorithm uses $5.03n$ bytes and is faster than any other tested algorithm. Only on a single file on a single machine 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 $\Theta(n^2 \log n)$ 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.

The C source code of all algorithms described in this paper, and the complete collection of test files, are publicly available on the web [21]. For our lightweight suffix sorting algorithm we provide a simple API which makes the construction of the suffix array as simple as calling two C procedures.

Finally, we point out that suffix sorting is a very active area of research. All algorithms described in this paper are less than four year old and new ones are under development: in addition to the already mentioned mkvtree package [16], we know of a new suffix sorting algorithm [26] which, like qsufsort, uses $8n$ space and runs in $O(n \log n)$ time in the worst case. Preliminary tests show that this new algorithm is roughly two times faster than qsufsort and slightly faster than ds2 [26].

Acknowledgments

We would like to thank Hideo Itoh, Tsai-Hsing Kao, Stefan Kurtz, Jesper Larsson, Kunihiko Sadakane, and Julian Seward for clarifying some details on their work and/or providing the source or executable code of their algorithms. We also would like to thank Giovanni Resta and Giorgio Vecchiocattivi for their technical assistance in the experimental work.

References

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

[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.

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

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

[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.

[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.

[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.

[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.

[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.

[10] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th ACM-SIAM Symposium on Discrete Algorithm*, 2003.

[11] 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.

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

[13] 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.

[14] Tsai-Hsing Kao. Improving suffix-array construction algorithms with applications. Master's thesis, Dept. of Computer Science, Gunma University, February 2001.

[15] 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.

[16] S. Kurtz. Mkvtree package (available upon request).

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

[18] 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.

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

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

[21] G. Manzini and P. Ferragina. Lightweight suffix sorting home page. `http://www.mfn.unipmn.it/~manzini/lightweight`.

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

[23] 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.

[24] J. Seward. The BZIP2 home page, 1997. `http://sources.redhat.com/bzip2`.

[25] J. Seward. On the performance of BWT sorting algorithms. In *DCC: Data Compression Conference*, pages 173–182. IEEE Computer Society TCC, 2000.

[26] P. Vo. Personal communication.