Trade off Between Compression and Search Times in Compact Suffix Array*

Veli Mäkinen

Department of Computer Science, P.O Box 26 (Teollisuuskatu 23),
FIN-00014 University of Helsinki, Finland.
Email: vmakinen@cs.helsinki.fi

Abstract. *Suffix array* is a widely used full-text index that allows fast searches on the text. It is constructed by sorting all suffixes of the text in the lexicographic order and storing pointers to the suffixes in this order. Binary search is used for fast searches on the suffix array. *Compact suffix array* is a compressed form of the suffix array that still allows binary searches, but the search times are also dependent on the compression. In this paper, we answer some open questions concerning the compact suffix array, and study practical issues, such as the trade off between compression and search times, and show how to reduce the space requirement of the construction. Experimental results are provided in comparison with other search methods. The results show that usually the size of a compact suffix array is less than twice the size of the text, while the search times are still comparable to those of suffix arrays.

1 Introduction

The classical problem in string-matching is to determine the occurrences of a short pattern P in a large text T. Usually the same text is queried several times with different patterns, and therefore it is worthwhile to preprocess the text in order to speed up the searches. Preprocessing builds an index structure for the text.

To allow fast searches for patterns of any size, the index must allow access to all suffixes of the text. This kind of indexes are called *full-text indexes*. Optimal inquiry time, which is $O(m + occ)$ as every character of P of length $m = |P|$ must be checked and the occ occurrences must be reported, can be achieved by using the *suffix tree* [15, 12, 14] as the index. In a suffix tree every suffix of the text is presented by a path from the root to a leaf.

The space requirement of a suffix tree is relatively high. It can be $12n$ bytes in the worst case even with a careful implementation [3], where $n = |T|$ is the size of T[1]. In addition, there is always an alphabet dependent factor on search times in any practical implementation.

* A work supported by the Academy of Finland under grant 22584.

[1] We assume an environment, where an integer takes 32 bits, and a byte takes 8 bits. The space requirements of the index structures do not include the n bytes for representing the text.

The *suffix array* [11, 5] is a reduced form of the suffix tree. It represents only the leaves of the suffix tree in the lexicographic order of the corresponding suffixes. A suffix array takes $4n$ bytes, and provides searching in $O(m \log n + occ)$ time with binary search. This can be further improved to $O(m + \log n + occ)$ by storing information about longest common prefixes (LCPs) [11].

There is often repetition in a suffix array such that some areas of the suffix array can be represented as links to the other areas. This observation results in a new full-text index called *compact suffix array* [13]. By controlling which areas are replaced by links, fast searches are still possible. This gives us a trade off problem, where the search times depend on the compression.

In this paper we consider two open problems on the compact suffix array. Namely, how to construct a minimal compact suffix array, and how to obtain the LCP information for the compact suffix array. We also analyze the space requirement of the construction, and show that a minimal compact suffix array can be constructed in $O(n \log n)$ time using at most $9n$ bytes (or $13\frac{1}{8}n$ bytes if the LCP values are calculated). Then we study experimentally the trade off problem between compression and search times, and show that a compact suffix array of size less than $2n$ can still be efficient in searches.

2 Compact Suffix Array

Let Σ be an ordered alphabet and $T = t_1 t_2 \ldots t_n \in \Sigma^*$ a text of length $n = |T|$. A *suffix* of text T is a substring $T_{i \ldots n} = t_i \ldots t_n$. A suffix can be identified by its *starting position* $i \in [1 \ldots n]$. A *prefix* of text T is a substring $T_{1 \ldots i}$. The length of the *longest common prefix* (LCP) of suffixes i and i' is $min\{l \mid T_{i+l} \neq T_{i'+l}\}$.

Definition 1 *The* suffix array *of text T of length $n = |T|$ is an array $A(1 \ldots n)$, such that it contains all starting positions of the suffixes of the text T such that $T_{A(1)\ldots n} < T_{A(2)\ldots n} < \ldots < T_{A(n)\ldots n}$, i.e. the array A gives the lexicographic order of all suffixes of the text T.*

A suffix array with *LCP information* is such that the LCPs between the suffixes that are consecutive in the *binary search order*, are stored as well. Suffixes are consecutive in the binary search order if a pattern is compared to them consecutively in binary search. Each suffix can have only two preceding suffixes in binary search order, and only the maximum of the two values needs to be stored together with indication of which value is stored (left or right). The other value can be calculated during the binary search [11]. An example of the suffix array with LCP information is shown in Fig. 1, where column *lcp* stores the LCP values (+ = left, - = right). With the LCP information, a pattern of length m can be searched in $O(m + \log n + occ)$ time, because each character of the pattern must be compared only once during the search.

The idea of compacting the suffix array is the following: Let $\Delta > 0$. Find two areas $a \ldots a + \Delta$ and $b \ldots b + \Delta$ of A that are repetitive in the sense that the suffixes represented by $a \ldots a + \Delta$ are obtained, in the same order, from the suffixes represented by $b \ldots b + \Delta$ by deleting the first symbol. In other words,

$A(a+i) = A(b+i)+1$ for $0 \le i \le \Delta$. Then replace the area $a \dots a+\Delta$ of A by a link, stored in $A(a)$, to the area $b \dots b+\Delta$. This operation is called a *compacting operation*. The areas may be compacted recursively, meaning that area $b \dots b+\Delta$ (or some parts of it) may also be replaced by a link. Therefore, we define an *uncompacting operation* to retrieve area $a \dots a+\Delta$ from an area $b \dots b+\Delta$ that is recursively compacted using a compacting operation. A path that is traversed when a suffix is uncompacted, is called a *link path*.

Due to the recursive definition, we need three values to represent a link:

- A pointer x to the entry that contains the start of the linked area.
- A value Δx such that entry $x+\Delta x$ denotes the actual starting point after entry x is uncompacted.
- The length of the linked area Δ.

Definition 2 *The* compact suffix array (CSA) *of text T of length $n = |T|$ is an array $CA(1 \dots n')$ of length n', $n' \le n$, such that for each* entry $1 \le i \le n'$, *$CA(i)$ is $(B, x, \Delta x, \Delta)$, where the value of $B \in \{\text{"}suffix\text{"}, \text{"}link\text{"}\}$ determines whether x denotes a suffix of the text, or whether $x, \Delta x$, and Δ denote a link to an area obtained by a compacting operation. The suffixes that are explicitly present in the structure are in the lexicographic order.*

The *optimal* CSA for T is such that its size n' is the smallest possible. We use $CA(i).B$, $CA(i).x$, $CA(i).\Delta x$, and $CA(i).\Delta$ to denote the value of B, x, Δx, and Δ at entry i of the array CA. An example of the CSA is shown in Fig. 1.

text:	g	t	a	g	t	a	a	a	c
pos:	1	2	3	4	5	6	7	8	9

suffix array:

entry	suffix	pos	lcp
1	aaac	6	-
2	aac	7	+2
3	ac	8	+1
4	agtaaac	3	+1
5	c	9	+0
6	gtaaac	4	-3
7	gtagtaaac	1	+0
8	taaac	5	-2
9	tagtaaac	2	-

➔

entry	suffix / link	pos / (x)
1	aaac	6
2	→	(1)
3	→	(2)
4	agtaaac	3
5	→	(3)
6	→	(4)
7	gtagtaaac	1
8	taaac	5
9	tagtaaac	2

➔

compact suffix array:

entry	suffix / link	pos / (x,Δx,Δ)	lcp
1	aaac	6	-
2	→	(1,0,2)	-
3	agtaaac	3	+1
4	→	(2,1,2)	-
5	gtagtaaac	1	+0
6	taaac	5	-2
7	tagtaaac	2	-

Fig. 1. Suffix array of text "*gtagtaaac*" and the construction of (binary searchable) compact suffix array.

3 Constructing an Optimal CSA from a Suffix Array

Compact suffix array can be constructed as follows: First, the areas that can be replaced by links are marked in the original array. This is done by advancing from the first entry of the array to the last, and finding for each suffix $A(i)$ the entry x that contains suffix $A(i) - 1$. The entry x of suffix $A(i) - 1$ can be found in constant time by constructing the reverse A_r of array A in advance; the array A_r is easily computed in linear time by setting $A_r(A(i).x) = i$ for all $1 \dots n$. After the entry x of suffix $A(i) - 1$ is found, the repetitiveness is checked between the areas starting at i and x, and in the positive case, the area starting at i is marked to be replaced by a link to the area starting at x. The second step is to construct the compacted array by replacing marked areas by links, and changing links to refer to the entries of the compact array. Both steps work in linear time. An example of the construction is shown in Fig. 1.

The above method was described in [13], but a heuristic was used to avoid *cyclic links*. Figure 1 shows a simple example of a cyclic link; suffixes 7 and 8 are replaced by links to suffixes 6 and 7, i.e. an area is linked to itself. In general, cyclic link paths can be generated. However, an area can not be replaced totally by itself via link paths; there is always one suffix where the uncompacting can start. In the next section, we will give an algorithm that can uncompact these kind of cyclic links. Therefore, we can remove the heuristic of [13] and state that the above algorithm constructs the optimal CSA.

4 Binary Search over CSA

4.1 Uncompacting Cyclic Links

The searches from the CSA can be done as from the suffix array, using binary search. To make the searches efficiently, we need the following modification to the definition of the CSA.

Definition 3 *The* binary searchable compact suffix array (BSCSA) *of text T is an array $CA(1 \dots n')$ which is as in the definition of the CSA with following limitations: $CA(1).B = CA(n').B =$ "$suffix$" and for each $i \in [2 \dots n' - 1]$ if $CA(i).B =$ "$link$" then $CA(i+1).B =$ "$suffix$".*

The algorithm in Sect. 3 that constructs the optimal CSA can be modified to construct the optimal BSCSA by just skipping one suffix each time a link is created. From this on, we assume that a CSA is a BSCSA (as in Fig. 1).

As only every other entry of the CSA can be a link, we can make binary searches over the entries that contain suffixes; if an entry containing a link is chosen to be the next left or right limit, we can choose its preceding entry instead. With two binary searches we can find entries l and r such that the pattern may occur in entries $l \dots r$. Then we can uncompact all the links in that area. If an entry l contains a link, the beginning of the pattern occurrences must be searched after uncompacting that link (with another binary search). Similarly the end of the pattern occurrences must be searched from the suffixes represented by the entry r. Now, it remains to define how the links can be uncompacted. In [13] a recursive procedure for this purpose was given under the assumption that no cyclic links exist. We now give an algorithm that can uncompact also cyclic links. The algorithm is shown in Fig. 2.

UnCompactLink(in CA,in x,in d,in out UCA,in out j,in Δr,in Δp):

$y \leftarrow CA(x).x$; $\Delta x \leftarrow CA(x).\Delta x + \Delta p$; $\Delta \leftarrow min(CA(x).\Delta - \Delta p, \Delta r)$

$j' \leftarrow j$; $i \leftarrow 0$

while $\Delta x > 0$ **do begin** {Skip links and suffixes that are not required.}

    if $CA(y+i).B = false$ **then begin**

        $i \leftarrow i+1$; $\Delta x \leftarrow \Delta x - 1$ **end**

    else begin

        if $CA(y+i).\Delta \leq \Delta x$ **then begin**

            $\Delta x \leftarrow \Delta x - CA(y+i).\Delta$; $i \leftarrow i+1$ **end**

        else break end end

while $j - j' < \Delta$ **do begin** {Uncompact until required suffixes are copied.}

    if $CA(y+i).B = "link"$ **then begin**

        UnCompactLink($CA, y+i, d+1, UCA, j, \Delta - (j - j'), \Delta x$)

        $\Delta x \leftarrow 0$ **end**

    else begin

        $UCA(j).x \leftarrow CA(y+i).x + d + 1$; $j \leftarrow j+1$ **end**

    $i \leftarrow i+1$ **end**

Fig. 2. An algorithm to uncompact a link.

The difference between this version of *UnCompactLink* and the version in [13] is that the information about the callers is carried through the recursion, in order to uncompact only those suffixes that are needed by the first caller. The ability to uncompact cyclic links follows, because no extra suffixes are uncompacted, and all the links that do not contain the required suffixes are skipped.

Theorem 4 *Function* UnCompactLink *works correctly, i.e. it returns an array* $UCA(1 \ldots \Delta)$ *that contains the suffixes that are represented by a link* $(x, \Delta x, \Delta)$ *in a compact suffix array* CA.

Proof. We use induction for the proof. Consider the recursive calls to the *UnCompactLink* as a tree, where the first caller is the *root* and each call to the *UnCompactLink* forms a *child node* in the tree, whose *parent node* is the caller. The tree is traversed in depth-first order. If the root passes parameters *UnCompactLink*(CA,x,0,UCA,1,$CA(x).\Delta$,0) to its only child, then the required information is available at the beginning. Assume that at some point in the depth-first traversal, the correct information is passed to a child.

Now, Δx gets the value of how many suffixes must be skipped in this sub-tree (the skip value passed from the parent plus the cumulated skip value in the path from root to the parent node), and Δ gets the value of how many suffixes must be uncompacted in this sub-tree (the minimum of the two values: (1) the length of the linked area passed from the parent minus the cumulated skip value in the path from root to the parent node, and (2) the amount of suffixes that are remaining to be uncompacted for the root). Lines (3)-(9) skip all the suffixes and links that can not represent the suffixes that are required for the root.

If the skipping stops at some link ($\Delta x > 0$ after line (9)), then that link is uncompacted at line (12) with parameters $\Delta r \leftarrow \Delta - (j - j') = \Delta$ and $\Delta p \leftarrow \Delta x$, which are correct, because Δr gets the value of how many suffixes are remaining to be uncompacted in this sub-tree, and Δp gets the value of how many suffixes should be skipped.

If the skipping stops with value $\Delta x = 0$ or Δx is set to zero at line (13), then the suffixes are uncompacted correctly at line (15) or links are uncompacted at line (12), until the condition at line (10) becomes unsatisfied (i.e. all required suffixes are uncompacted). The correctness of the call at line (12) follows, because the parameters get the right values; especially $\Delta r \leftarrow \Delta - (j - j')$ (the amount of required suffixes minus already uncompacted suffixes) and $\Delta p \leftarrow \Delta x = 0$ (no skipping). Finally, the value of j is updated during the uncompacting, and the parent gets the information of how many suffixes were uncompacted by its child. □

4.2 Constructing LCP Values to Speed Up Searches

The LCP information can be used to speed up the searches in the CSA similarly as in the suffix array. The LCP values are only needed for the suffixes that are present in the CSA, and so these values can be stored in the same space that is allocated for the fields Δx and Δ. The LCP values for the CSA can be obtained in linear time as follows:

(1) Use the suffix array construction algorithm of [11] to construct the LCP values between the adjacent suffixes in the suffix array.
(2) Copy the LCP values of adjacent suffixes from suffix array to the suffixes that are present in the CSA, modifying the LCP values of the suffixes that follow a linked area so that such LCP value is the minimum of the LCP values of the suffixes represented by the preceding link and the suffix itself.
(3) From the LCP values of adjacent suffixes, obtain the LCP values of the suffixes that are consecutive in the binary search order as in [11].

There is one drawback in the above method; the fastest suffix array construction algorithms do not provide the LCP values (see Larsson and Sadakane [10]). Kasai *et. al.* [8] gave an algorithm to construct the LCP values from the suffix array in linear time, but their algorithm uses more space than an efficient suffix array construction. We give here a more space-efficient version of the algorithm (Fig. 3). The space-efficiency is achieved by using an array in two purposes and an algorithm to reverse a suffix array in linear time using only n extra bits.

Reverse(in out SA, in n):
(1) **for each** $i \in [1 \ldots n]$ **do begin**
(2) $\quad j \leftarrow SA(i)$; $x \leftarrow i$
(3) $\quad$ **while** state$(j) =$ false **do begin**
(4) $\quad\quad j' \leftarrow j$; $j \leftarrow SA(j')$; $SA(j') \leftarrow x$; state$(j') \leftarrow$ true; $x \leftarrow j'$ **end end**

SAtoLCP(in T, in SA, out LCP):
(1) $n \leftarrow |T|$; Copy(SA, LCP); Reverse(LCP, n); $last \leftarrow 0$; $new \leftarrow 0$
(2) **for each** $i \in [1 \ldots n]$ **do begin**
(3) $\quad i' \leftarrow LCP(i)$
(4) $\quad$ **if** $i' = 1$ **then** $LCP(i) \leftarrow 0$
(5) $\quad$ **else begin**
(6) $\quad\quad last \leftarrow max(0, last - 1)$
(7) $\quad\quad new \leftarrow$ CalcLCP$(T_{SA(i'-1)+last \ldots n}, T_{SA(i')+last \ldots n})$
(8) $\quad\quad LCP(i) \leftarrow last + new - 1$ **end**
(9) $\quad last \leftarrow LCP(i)$ **end**
(10) Reverse(SA, n); Reverse(SA, LCP, n)

Fig. 3. Algorithms for reversing a suffix array and calculating the LCP values between adjacent suffixes in the suffix array. The call to function *CalcLCP(A,B)* returns the length of the LCPs between strings A and B. The call to *Reverse(SA,n)* with an additional parameter LCP, reverses the LCP array in the same order as SA is reversed.

The algorithm *Reverse* in Fig. 3 reverses a suffix array as follows: the value of i is written in the entry $SA(i)$ and the old content denotes the next entry where the value $SA(i)$ is written, and so on, until an entry is reached which has already been updated. This is repeated for each i. The algorithm works also in the other direction (from reverse array to original array).

The algorithm *SAtoLCP* in Fig. 3 calculates the LCP values. First, it copies the reverse of the suffix array into LCP array, and calculates the LCP values from suffix 1 to suffix n. After line (9) the LCP values are stored in reverse order, and so the array LCP is reversed at line (10).

The linear time requirement of algorithm *SAtoLCP* follows from the fact that the comparisons done with the preceding suffix are used in the next comparison. Let $lcp(i)$ denote the length of the LCP between suffixes $i = SA(x)$ and $SA(x-1)$ in the suffix array SA of length n, when $1 < x \leq n$, and let $lcp(i) = 0$, when $i = SA(1)$. Now, the sum of the comparisons is $lcp(1)+1+lcp(2)-max(0, lcp(1)-1)+1+\cdots+lcp(n)-max(0, lcp(n-1)-1)+1 \leq 2n$, where values $+1$ are due to the comparison to the first mismatching character.

Theorem 5 *Given a text T of length n and its suffix array, there is an algorithm (Fig. 3) to construct the LCP information between the adjacent suffixes in suffix array in $O(n)$ time using $n + O(1)$ bits additional work space.* □

4.3 Limiting Compression to Speed Up Searches

In the worst case, the searches in the CSA take linear time, because a link may represent $O(n)$ suffixes or the length of a link path to a suffix may be $O(n)$. To provide a guarantee to the search times, we must limit both the lengths of the linked areas, and the lengths of the link paths. We introduce constants C and D for these purposes. In [13], it was proven that a link can be uncompacted in $O(C^2 D')$ time, where $D' = max(1, D - \log C)$. Although we have an improved version of function *UnCompactLink* (Fig. 2), this bound is still valid.

Now, we have the following search times for the (C, D) limited CSA: To report $occ > 0$ occurrences of pattern of length m in text of length n takes $O(m \log C + \log n + C^2 D' + occ\, C\, D')$ time in the worst case. To get the time needed for *existence queries* (to determine whether $occ > 0$), set $occ = 0$.

To ease the comparison to other index structures, we can choose $D = \lceil \log \lceil \log n \rceil \rceil$ and $C = 2^D - 1$. Now, it takes $O(m \log \log n + \log^2 n + occ \log n)$ time to report $occ > 0$ occurrences (set $occ = 0$ for existence queries).

The average case analysis was done in [13] under the assumption that all the areas in the suffix array are linked equally likely (note that this assumption follows from assuming uniform Bernoulli model, but is more general). With this assumption, the average values for the length of the linked areas and for the length of the link paths are both $\frac{2n-n'}{n'}$, where n' is the size of the CSA. Now we have $O(m \log \frac{2n-n'}{n'} + \log n + (\frac{2n-n'}{n'})^2 + occ \frac{2n-n'}{n'})$ search time for reporting $occ > 0$ occurrences (set $occ = 0$ for existence queries).

5 Space requirement

5.1 Minimal CSA

An entry of the CSA consists of three integers and one boolean value. Unless the text is very compressible, the CSA would occupy more space than the suffix array. To guarantee a limit for the search times, we already introduced constants C and D to limit the length of linked areas and link paths. It follows that values Δx and Δ can be represented in $\lceil \log(C+1) \rceil$ bits. Also value x can be represented in $\lceil \log n \rceil$ bits. Therefore the CSA can be stored as four bit vectors of sizes n' (values B), $n' \lceil \log n \rceil$ (values x), and $n' \lceil \log(C+1) \rceil$ (values Δx and Δ), while the access to each entry is guaranteed in constant time using bit operations. If the LCP values are not needed, fields Δx and Δ are unused in the entries containing suffixes, and we can use the same array to store both values (if an entry i contains a link, then the entry $i + 1$ contains a suffix, and we can store value Δx in the entry i and value Δ in the entry $i + 1$).

To get the most space efficient CSA (i.e. *minimal CSA*), we can simulate the construction algorithm with values $C \in \{2^0 - 1, 2^1 - 1, \ldots, 2^{\lceil \log n \rceil} - 1\}$ $(D = \infty)$ and choose the value that results into the minimal CSA. Therefore we can construct the minimal CSA in $O(n \log n)$ time. A minimal CSA takes at most the same space as a suffix array, because with value $C = 0$ the minimal CSA can be represented as a suffix array.

5.2 Reducing the Space Requirement of the Construction

A straightforward implementation of the construction algorithm in Sect. 3 takes $16n$ bytes. We use similar technique as in calculating the LCP values in Fig. 3 to reduce the space requirement of the construction; the same array is used in two purposes. At the beginning, an array is used to store the reverse of the suffix array to provide constant access to locate suffixes. At the end, the same array contains the suffix array, in which the linked areas are marked. The same technique is used again to compact the marked array.

The optimal CSA can be constructed using only $8n$ bytes. To construct the minimal CSA or a CSA that provides guaranteed worst case limits for search times, we have to include values C and D to the construction algorithm.

Limiting the lengths of the linked areas with value C is easy; we can just break long linked areas into smaller parts. Limiting the lengths of the link paths is more complicated, but it can be done by introducing a new array $pathdepth(1 \ldots n)$ to store path depths for each suffix. Updating the array $pathdepth$ can be done in constant time by storing the path depths for both at the begin and at the end of the link paths. When two paths are combined, the length of the combined paths can be checked and updated in constant time.

Because the use of the value D is only worthwhile when the value is small, we can assume that D fits into a byte. Therefore the whole construction uses $9n$ bytes. When LCP values are needed, the construction can still be done using $13\frac{1}{8}n$ bytes, by using the algorithms presented in Sect. 4.2.

6 Experimental results

We have implemented all the methods discussed in this paper. For testing purposes it would have been enough to simulate most of the algorithms using easier implementations, but as we wanted to test with large datasets, the more space efficient implementations became necessary. However, the space efficient implementations did have impact on the running times; especially the use of arrays with given length bit fields (other than byte, etc.), slowed down the processing by a factor of circa 1.5.

The test-program was written in C++, and compiled with g++ version 2.7.2.3 with option -O3 in Linux 2.0.38 environment. The tests were run on a 700 MHz Pentium III machine with 768 MB RAM. The running times were measured in milliseconds using *times* library function.

We used a collection of different size text files and a DNA-sequence (Human chromosome 7) in our experiments (see Table 1). The DNA-sequence (68 MB) was almost the largest file that could be indexed in our machine; its compact suffix array could be constructed in the RAM, but for the LCP values we had to use the trivial $O(n \log n)$ expected case ($O(n^2)$ worst case) algorithm, because the $O(n)$ algorithm required too much space ($13\frac{1}{8}$*68 MB>768 MB).

There are basically three parameters that control the efficiency of the CSA index structure. Values C (maximum length of a linked area) and D (maximum

Table 1. The files that were used in tests are listed below. Some files (marked as "c" in table) are from Canterbury and Calgary corpora (http://corpus.canterbury.ac.nz), and others (marked as "g") are from Project Gutenberg (http://www.promo.net/pg). To simulate a large text file, we catenated files marked as "1" to a file named *txtcorpus*. The other large file is a DNA-sequence of Human chromosome 7 (the *.lbr* in the end of *07hgp10.txt* means that line breaks are removed from the original file).

text	size	corpus	type	0/1	text	size	corpus	type	0/1
alice29.txt	152 089	c	text	1	pge0112.txt	8 588 144	g	text	1
bible11.txt	5 073 934	g	text	1	plrabn12.txt	481 861	c	text	1
book1	768 771	c	text	0	progp	49 379	c	code	0
crsto10.txt	2 751 761	g	text	1	shaks12.txt	5 582 655	g	text	1
kjv10.txt	4 545 377	g	text	1	suntx10.txt	79 787	g	text	1
lcet10.txt	426 754	c	text	1	sunzu10.txt	344 157	g	text	1
nkrnn09.txt	2 061 602	g	text	1	taofj10.txt	3 059 101	g	text	1
nkrnn10.txt	2 062 563	g	text	1	txtcorpus	43 930 831		text	0
nkrnn11.txt	2 075 075	g	text	1	world192.txt	2 473 400	c	text	1
paper1	53 161	c	text	0	07hgp10.txt.lbr	71 003 068	g	DNA	0

length of a link path), and the use of the LCP values, have all effect on both the compression and the search times.

We first examine the effect of the values C and D. We ran two tests varying first the values of C and then D on text *txtcorpus*. The results are shown in Fig. 4. Instead of measuring time, we calculated the amount of operations that were needed to uncompact the whole array[2]. It can be seen that both parameters affect the uncompacting time considerably, although the behavior is not as bad as what could be expected from the worst case analysis. On the other hand, the average case analysis is too optimistic, because it is based on the assumption that each entry is equally likely to be the start of a linked area, which is not true for usual texts. Typically some areas of the text are more compressible than others (like white space sequences), and therefore there are areas in a compact suffix array where uncompacting is closer to worst case behavior.

The compression was not improved significantly as C and D grew except when they were small. Usually the minimal CSA is achieved with small C (between 7 and 63 in our test set), and therefore we used minimal CSAs in the remaining tests, and controlled the trade off between compression and search times by choosing an appropriate value for D.

Next we studied the effect of the LCP values. We used three sets of patterns: small (4-10), moderate (30-50), and large (200-500). Each set consisted of 10000 patterns, selected randomly from the files in Table 1 (separate sets for text files and DNA). The results are shown in Table 2.

[2] Actually, the whole array could be uncompacted in linear time if needed; we can store the suffixes represented by a link in their places in a suffix array at the first time we uncompact that link. Before uncompacting a link, we can check if it already has been uncompacted. Thus, each suffix can be uncompacted in constant time.

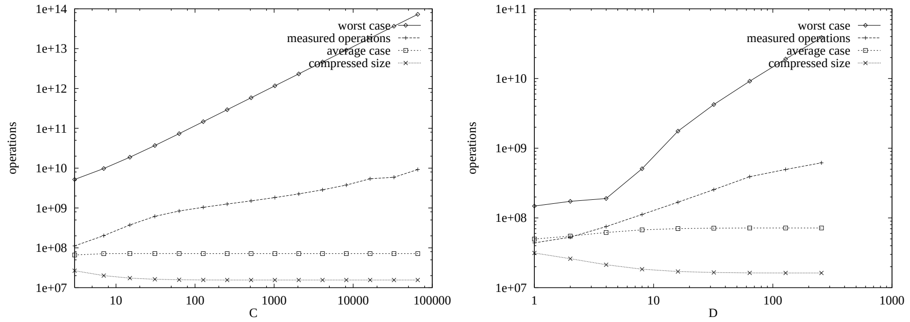

Fig. 4. The effect of values C (left) and D (right) on uncompacting the CSA of file *txtcorpus*. In the left, the values of C are $2^2 - 1, 2^3 - 1, \ldots, 2^{16} - 1$, while $D = 256$. In the right, the values of D are $2^1, 2^2, \ldots, 2^8$, while $C = 31$. The worst case behaviour was calculated with formula $C^2(D - \log C)l + s$, where l is the amount of links in the CSA and s is the amount of suffixes in the CSA (i.e. $l + s = n'$). The average case behaviour was calculated similarly.

The LCP values did not provide any significant speed up in our tests. In fact, Table 2 shows that searching was usually slower with LCP values; the more complicated implementation seems to overrule the effect of LCP values, especially because accessing the array fields was much more costly than in standard array implementation. Similar behaviour dominated also the searches from the suffix array. To really see the effect of LCP values, we tested also with such an extreme text as $a^{1000000}$ and pattern a^{1000}, where searching was 19 times faster with LCP values in suffix array.

Finally, we compared CSA with suffix array and sequential search. In sequential search, we used the implementation of Boyer-Moore-Horspool algorithm [7] from [4] (with a simple modification to report all the matches). We used the algorithm of Larsson and Sadakane [10] to construct suffix arrays. To guarantee a fair comparison, we represented suffix arrays in $n\lceil \log n \rceil$ bits instead of the usual $4n$ bytes implementation. We did not use LCP values in comparison, because they did not provide any speed up in practise. The results are shown in Table 3.

The results in Table 3 show that compact suffix arrays take much less space than suffix arrays, and the penalty in search times is acceptable if there are not too many occurrences to report. The removal of the heuristic to avoid cyclic links, is crucial (compare the values in column $|CSA1|$). The gap between sequential search and indexed search is huge, in favor of the index.

7 Discussion

The large space requirement of full-text indexes is often the reason why they cannot be used in applications in which they would otherwise be a natural solution. We have shown that compact suffix arrays can occupy less than two times

Table 2. The effect of the LCP values. We measured the overall time to search (not report) 10000 patterns averaged over 100 trials for three sets consisting of different size patterns. Three different versions of the CSA were used: CSA is without LCP values, CSA_LCP is with LCP values, and CSA_LCPfit is with LCP values that fit into the space of fields Δx and Δ.

	Parameters			Search times (ms)		
text	|pattern|	C	D	CSA	CSA_LCP	CSA_LCPfit
txtcorpus	4-10	31	12	107	130	136
txtcorpus	30-50	31	12	164	180	188
txtcorpus	200-500	31	12	224	234	242
07hgp10.txt.lbr	4-10	7	12	79	96	92
07hgp10.txt.lbr	30-50	7	12	116	121	116
07hgp10.txt.lbr	200-500	7	12	187	176	168

Table 3. The comparison of search times and space requirement. We measured the overall time to search and report 10000 patterns averaged over 100 trials (except only one trial for sequential search). Small patterns were used for texts, and moderate for DNA. Column *occ* denotes either the amount of patterns found (upper line on each text) or the amount of occurrences (lower line on each text). CSA1 denotes the minimal CSA with $D = 12$, CSA2 denotes CSA with $D = \lceil \log \lceil \log n \rceil \rceil$ and $C = 2^D - 1$, SA denotes the suffix array, and BMH denotes the Boyer-Moore-Horspool search. The values in parentheses in column $|CSA1|$ denote the space requirement when the cycles were prevented with the heuristic of [13].

	Index size / text size			occ	Search times (ms)			
text	|CSA1|	|CSA2|	|SA|		CSA1	CSA2	SA	BMH
book1	1.71	1.83	2.50	996	65	66	36	43020
book1	(2.13)			5176	67	68	36	46760
paper1	1.29	1.46	2.00	152	52	46	25	1340
paper1	(1.69)			346	51	47	25	1550
pge0112.txt	1.75	1.90	3.00	6173	85	80	48	312580
pge0112.txt	(2.36)			98557	133	116	49	520300
progp	1.01	1.26	2.00	14	54	44	23	1190
progp	(1.38)			34	54	44	24	1370
txtcorpus	1.59	1.84	3.25	10000	106	91	56	1384570
txtcorpus	(2.30)			351354	301	212	60	2647240
world192.txt	1.15	1.41	2.75	1700	131	85	42	130850
world192.txt	(1.81)			35368	150	96	42	148160
07hgp10.txt.lbr	2.68	2.87	3.38	10000	115	118	91	4038640
07hgp10.txt.lbr	(2.98)			14542	124	124	91	8571530

the size of the text with search times comparable to those of suffix arrays. Also, compact suffix arrays can be constructed asymptotically as fast as suffix arrays, with similar space requirements at construction time.

Still, compact suffix arrays do take more space than the text itself (the worst case is $O(n \log n)$ bits as in suffix arrays), making them too expensive in many applications. There is also recent work considering full-text indexes that occupy $O(n)$ bits [9, 6], or even less [1]. They also have some penalty on search times, but the experimental results [2] considering the opportunistic index of [1] show that the index is quite competitive; an indirect comparison to compact suffix array with text *world192.txt* (through comparing relative performance to suffix array) show that counting the amount of pattern occurrences is equally fast, and reporting the occurrences is 10 times slower than from compact suffix array, while the opportunistic index takes 7 times less space than the compact suffix array.

References

1. P. Ferragina and G. Manzini, *Opportunistic Data Structures with Applications*, In Proc. IEEE Symposium on Foundations of Computer Science, 2000.
2. P. Ferragina and G. Manzini, *An Experimental Study of an Opportunistic Index*, In Proc. ACM-SIAM Symposium on Discrete Algorithms, 2001, to appear.
3. R. Giegerich, S. Kurtz, and J. Stoye, *Efficient Implementation of Lazy Suffix Trees*, In Proc. Third Workshop on Algorithmic Engineering (WAE99), LNCS 1668, Springer Verlag, 1999, pp. 30-42.
4. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures - In Pascal and C*, Addison-Wesley, Wokingham, UK, 1991. (second edition).
5. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, *Lexicographical indices for text: Inverted files vs. PAT trees*, Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.
6. R. Grossi and J. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, In Proc. 32nd ACM Symposium on Theory of Computing, 2000, pp. 397-406.
7. R. N. Horspool, *Practical fast searching in strings*, Soft. Pract. and Exp., 10, 1980, pp. 501-506.
8. T. Kasai, H. Arimura, and S. Arikawa, *Virtual suffix trees: Fast computation of subword frequency using suffix arrays*, In Proc. 1999 Winter LA Symposium, 1999, in Japanese.
9. J. Kärkkäinen, *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.
10. N. Jesper Larsson and K. Sadakane, *Faster Suffix Sorting*, Technical Report, number LU-CS-TR:99-214, Department of Computer Science, Lund University, Sweden, 1999.
11. U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput., 22, 1993, pp. 935-948.
12. E. M. McCreight, *A space economical suffix tree construction algorithm*, Journal of the ACM, 23, 1976, pp. 262-272.
13. V. Mäkinen, *Compact Suffix Array*, In Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000), LNCS 1848, 2000, pp. 305-319.
14. E. Ukkonen, *On-line construction of suffix-trees*, Algorithmica, 14, 1995, pp. 249-260.
15. P. Weiner, *Linear pattern matching algorithms*, In Proc. IEEE 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1-11.