Fundamenta Informaticae Special Issue – Computing Pattern in Strings, 56(1-2), 2003, pp. 191–210 191
IOS Press

Compact Suffix Array — A Space-Efficient Full-Text Index

Veli Mäkinen*
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
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 give efficient methods for constructing and querying compact suffix arrays. We also 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. With a large text corpora, the index took 1.6 times the size of the text, while the searches were only two times slower than from a suffix array.

Keywords: Suffix array, index compression, full-text index, string matching

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 $O(m + \mathrm{occ})$ as every character of P of length $m = |P|$ must be checked and the occ occurrences must be reported, can be

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

achieved by using the *suffix tree* [27, 19, 26] 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 [8], 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.

The *suffix array* [18, 10] 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 $O(m \log n + \mathrm{occ})$ time with binary search. This can be further improved to $O(m + \log n + \mathrm{occ})$ by storing information about longest common prefixes (LCPs) [18].

A compressed version of the suffix tree is obtained by combining its similar subtrees; the result is a *Compact Directed Acyclic Word Graph (CDAWG)* [1, 2, 4, 5]. We propose a compressed version of the suffix array called *compact suffix array (CSA)*. The CSA can be considered as an analogue to the CDAWG; as the CDAWG is a minimized suffix tree, the CSA is a minimized suffix array. Despite the analogue, these two structures are not in one-to-one correspondence to each other, because there is different kind of repetitiveness in the suffix tree than in the suffix array due to different data representation.

A compact suffix array is obtained by replacing some areas of a suffix array with links to the other areas. 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 give efficient algorithms for constructing compact suffix arrays; we show that compact suffix arrays can be constructed in linear time from suffix arrays. Then we give an efficient search algorithm and analyze its running time both in the worst and in the expected case. 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.

This full paper integrates and complements the results given in two conference papers [20, 21].

2. Compact Suffix Array

Let Σ be an ordered alphabet and $\epsilon \notin \Sigma$ an empty symbol smaller than any symbol in Σ. Let $T = t_1 t_2 \dots t_n \in \Sigma^*$ be a text of length $n = |T|$. A *suffix* of text T is a substring $T_{i \dots n} = t_i \dots t_n$. A suffix can be identified by its *starting position* $i \in [1 \dots n]$. A *prefix* of text T is a substring $T_{1 \dots i}$. The length of the *longest common prefix* (LCP) of suffixes i and i' is $\mathrm{lcp}(i, i') = \min\{l \mid t_{i+l} \neq t_{i'+l}, l \geq 0\}$. The comparison operations for strings $A, B \in \Sigma^*$ are defined as follows: $A < B$ (resp. $A > B$) if $a_i = b_i$ for all $1 \leq i < l$ and $a_l < b_l$ ($a_l > b_l$), where the strings A and B are augmented to the same length by adding symbols ϵ to the end of the shorter string.

Definition 2.1. The *suffix array* of text T of length $n = |T|$ is an array $A(1 \dots n)$, such that it contains all starting positions of the suffixes of the text T such that $T_{A(1) \dots n} < T_{A(2) \dots n} < \dots < T_{A(n) \dots n}$, i.e. the array A gives the lexicographic order of all suffixes of the text T.

[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.

A suffix array with *LCP information* is such that the lengths of 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 a binary search. Each suffix can have only two preceding suffixes in the 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 [18]. 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 $O(m + \log n + \mathrm{occ})$ time, because each character of the pattern must be compared only once during the search. We refer to [18] for more details about the use of the LCP values.

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 $0 \leq i \leq \Delta$. Then replace the area $a \ldots a + \Delta$ of A by a link, stored in $A(a)$, to the area $b \ldots b + \Delta$. This operation is called a *compacting operation*. The areas may be compacted recursively, meaning that area $b \ldots b + \Delta$ (or some parts of it) may also be replaced by a link. Therefore, we define an *uncompacting operation* to retrieve area $a \ldots a + \Delta$ from an area $b \ldots b + \Delta$ 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 2.2. A *compact suffix array (CSA)* of text T of length $n = |T|$ is an array $\mathrm{CA}(1 \ldots n')$ of length n', $n' \leq n$, such that for each entry $1 \leq i \leq n'$, $\mathrm{CA}(i)$ is $(B, x, \Delta x, \Delta)$, where the value of $B \in \{\text{“suffix”}, \text{“link”}\}$ 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. When all the links of CA are uncompacted, the result is the suffix array of text T.

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

As such, it is hard to make efficient searches over the CSA. To enable binary searches over the CSA and a sensible worst case analysis of the search method, we make the following modifications to the definition:

Definition 2.3. A $(\mathcal{C}, \mathcal{D})$ *-limited binary searchable compact suffix array (*$(\mathcal{C}, \mathcal{D})$ *-limited BSCSA)* of text T is an array $\mathrm{CA}(1 \ldots n')$ which is as in the definition of the CSA with the following limitations:

(1) $\mathrm{CA}(1).B = \mathrm{CA}(n').B =$ “suffix” and for each $i \in [2 \ldots n' - 1]$ if $\mathrm{CA}(i).B =$ “link” then $\mathrm{CA}(i + 1).B =$ “suffix”.

(2) The length of each linked area is at most $\mathcal{C}$, i.e. $\mathrm{CA}(i).\Delta \leq \mathcal{C}$ for all $1 \leq i \leq n'$.

(3) The length of each link path is at most $\mathcal{D}$.

In the sequel, we often omit repeating term "binary searchable", when it is clear from the context. An example of a 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	-

Figure 1. Suffix array of text "gtagtaaac" and the construction of a (binary searchable) compact suffix array. Note that the LCP values correspond to the suffixes that are consecutive in the binary search order: e.g. $\mathrm{lcp}[8] = -2$ in suffix array, since "taaac" will be compared with "gtagtaaac" and "tagtaac" in the binary search order. The latter comparison yields LCP of length 2 (and $-$ codes the fact that the latter comparison gives the largest LCP value).

2.1. Where Does the Compression Come From?

The compression achieved by the CSA is related to the amount of *suffix-free strings* in the text. A string $p = p_1 p_2 \ldots p_m$ is suffix-free if $p' = p_2 p_3 \ldots p_m$ does not occur anywhere else in the text but together with p. In a suffix array, the occurrences of p form a consecutive region as well as the occurrences of p'. Thus, the occurrences of p' can be replaced by a link to the occurrences of p. For an example, a string $p =$ *"procedure"* is suffix-free, if $p' =$ *"rocedure"* does not appear anywhere else but within the occurrences of p. This would most probably be the case when the text is a *Pascal*-program.

The connection to an efficient text compression method is now easy to give. In Burrows-Wheeler transformation [3], first the suffix array of the text is constructed. Then a transformed text is constructed by including the preceding character of each suffix in order of appearance in the suffix array. This transformed text is highly compressible, and more importantly, it can be decoded into the original text (see [3] for details). Much of the compressibility is due to the suffix-free strings. For an example, suffixes that start with *"rocedure"* form a continuous region in the suffix array. They all have preceding character *"p"*. Thus, there is a run of characters *"p"* in the Burrows-Wheeler transformed text, which can be efficiently encoded using run-length compression (i.e. using coding (*"p"*, length of the run)). These runs correspond to the linked areas in the CSA.

3. Constructing CSA from Suffix Array

We shall next describe how compact suffix arrays can be constructed in linear time given the suffix array of the text. We shall also describe how LCP values can be computed for a CSA.

3.1. Constructing Optimal CSA

An optimal 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 $1 \ldots n$. 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. The pseudo-code of the algorithm is shown in Fig. 2.

It easy to see that the algorithm produces an optimal compact suffix array; lines (6-7) in *MarkLinksTo-SuffixArray* guarantee that maximally long linked areas are formed. In other words, since the areas that are replaced by links do not overlap, the optimal solution is simply formed by greedily maximizing the sizes of areas to be replaced (i.e. any compact suffix array can be converted into an optimal one by extending its linked areas as much as possible).

Proposition 3.1. Given a suffix array of text of length n, an optimal CSA can be constructed in $O(n)$ time.

3.1.1. Constructing Other Versions of CSA.

The algorithm in Fig. 2 generalizes to constructing an optimal BSCSA in linear time; each time a link is made, the next suffix is skipped (use $i \leftarrow i + \Delta + 2$ at line (10) in *MarkLinksToSuffixArray*). An example of the execution of this version of the algorithm is given in Fig. 1. To see that the algorithm constructs an optimal BSCSA, consider the following. Let A be a BSCSA. To make A optimal, one can use almost the same principle as with the normal CSA; extend the linked areas as much as possible under the constraint that no two linked areas can become adjacent (one suffix must be left between two areas). This is a deterministic scheme except for the choice of the delimiting suffix between two areas. Let l and r be two areas that could become adjacent if l would be extended one suffix to the right or r would be extended one suffix to the left. Not depending on the choice made here (we chose the latter), the same amount of indices are needed in the optimal solution.

An optimal $(\mathcal{C}, \infty)$-limited (binary searchable) CSA can be constructed in a similar fashion; the long linked areas can be cut greedily into pieces of length at most $\mathcal{C}$. The optimality follows from the fact that the same amount of indices are needed in the solution no matter how the linked areas are cut into pieces.

This greedy procedure does not anymore extend to constructing an optimal $(\mathcal{C}, \mathcal{D})$-limited (binary searchable) CSA. This is because the way the long link paths are cut into pieces of length at most $\mathcal{D}$ affect the optimal choice of splitting long linked areas into pieces of length $\mathcal{C}$, and vice versa. In fact, we have

MarkLinksToSuffixArray(in A,in n,out A',out n'):
(1) **for each** $1 \leq i \leq n$ **do** $A'(i).x \leftarrow -1$
(2) **for each** $1 \leq i \leq n$ **do** $A_r(A(i).x) \leftarrow i$
(3) $i \leftarrow 1$; $n' \leftarrow 0$
(4) **while** $i \leq n$ **do begin**
(5) $x \leftarrow A_r(A(i).x - 1)$
(6) Let Δ be largest integer such that
(7) $A(i+j).x = A(x+j).x + 1$ for all $0 \leq j \leq \Delta$
(8) **if** $\Delta > 0$ **then begin** {Mark linked area}
(9) $A'(i+j) \leftarrow$ ("link", $x, j, \Delta + 1$) for all $0 \leq j \leq \Delta$
{If a link points to $i+j$, then $A'(i+j).\Delta x$ stores the skip value needed at line (9) in *CompactMarkedArray*}
(10) $i \leftarrow i + \Delta + 1$; $n' \leftarrow n' + 1$; **end**
(11) **else begin** {Copy suffix.}
(12) $A'(i) \leftarrow$ ("suffix", $A(i).x, 0, 0$)
(13) $i \leftarrow i + 1$; $n' \leftarrow n' + 1$; **end**; **end**

CompactMarkedArray(in A',in n,in n',out CA):
(1) $j \leftarrow 1$
(2) **for each** $1 \leq i \leq n$ **do begin**
(3) **if** $A'(i).\Delta x = 0$ **then begin** {Suffix or start of a linked area.}
(4) $\mathrm{CA}(j) \leftarrow A'(i)$
(5) mapiToj(i) $\leftarrow j$
(6) $j \leftarrow j + 1$; **end**
(7) **else** mapiToj(i) $\leftarrow j - 1$ **end** {Inside a link.}
(8) **for each** $1 \leq i \leq n'$ **such that** $\mathrm{CA}(i).B =$ "link" **do**
(9) $\mathrm{CA}(i) \leftarrow$ ("link",mapiToj($\mathrm{CA}(i).x$), $A'(\mathrm{CA}(i).x).\Delta x, \mathrm{CA}(i).\Delta$)

Figure 2. Algorithm that constructs an optimal CSA. The construction works in two phases: First, the suffix array A is given as an input to *MarkLinksToSuffixArray*, which outputs a marked array A'. Next, the marked array A' is given as an input to *CompactMarkedArray*, which then outputs the compact suffix array CA. For simplicity, arrays A and A' use the same representation as array CA.

already a problem in modifying the linear time greedy algorithm to construct a feasible $(\mathcal{C}, \mathcal{D})$-limited CSA in the first place. We will next describe the required modifications to the algorithm in Fig. 2.

To construct a $(\mathcal{C}, \mathcal{D})$-limited CSA we need some way of limiting the lengths of the link paths. This can be done as follows. Keep an array $d(1 \ldots n)$, where the lengths of the link paths are to be stored for each suffix during the construction. The array can be used to test whether two link paths can be joined so that their overall length will not exceed $\mathcal{D}$. If they can be joined, the length of the new link path can be updated in constant time for the suffixes in the begin and in the end of the new link path. For other suffixes this value need not to be updated, since the link paths grow only from the tail or from the head. More precisely, we need to replace lines (6) and (7) in *MarkLinksToSuffixArray* with

(6') Let Δ be the largest integer smaller than $\mathcal{C}$ such that
(7') $A(i+j).x = A(x+j).x + 1$ and $d(A(i+j).x) + d(A(x+j).x) + 1 \leq \mathcal{D}$ for all $0 \leq j \leq \Delta$

To update the lengths of the link paths, we need to add before line (9) in *MarkLinksToSuffixArray* the following two lines:

($8\frac{1}{3}$) $d(A(i+j).x + d(A(i+j).x)) = d(A(i+j).x) + d(A(x+j).x) + 1$ for all $0 \leq j \leq \Delta$
($8\frac{2}{3}$) $d(A(x+j).x - d(A(x+j).x)) = d(A(i+j).x) + d(A(x+j).x) + 1$ for all $0 \leq j \leq \Delta$

For notational reasons, we have used a slightly inaccurate formulation above. Actually, the updates of the link path lengths of earlier suffixes inside the same linked area may affect the link path lengths of the later suffixes. Thus, one must alternate the comparisons and updates so that the array d is already updated for values $j' < j$ before comparison $d(A(i+j).x) + d(A(x+j).x) + 1 \leq \mathcal{D}$ at line (7'). This can be easily implemented so that the time complexity remains linear.

3.2. 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 [18] to construct the LCP values between the adjacent suffixes in the suffix array.

(2) Copy the LCP values of adjacent suffixes from the 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 [18].

There is one drawback in the above method; the fastest suffix array construction algorithms do not provide the LCP values (see Larsson and Sadakane [17]). Kasai et al. [14] 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

```
Reverse(in out A, in n):
(1)  for each i ∈ [1 ... n] do begin
(2)      j ← A(i); x ← i
(3)      while state(j) = "false" do begin
(4)          j' ← j; j ← A(j'); A(j') ← x; state(j') ← "true"; x ← j' end end

SAtoLCP(in T, in A, out lcp):
(1)  n ← |T|; Copy(A, lcp); Reverse(lcp, n); last ← 0; new ← 0;
(2)  for each i ∈ [1 ... n] do begin
(3)      i' ← lcp(i)
(4)      if i' = 1 then lcp(i) ← 0
(5)      else begin
(6)          last ← max(0, last − 1)
(7)          new ← CalcLCP(T_{A(i'−1)+last...n}, T_{A(i')+last...n})
(8)          lcp(i) ← last + new end
(9)      last ← lcp(i) end
(10) Reverse(A, n); Reverse(A, lcp, n)
```

Figure 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(*A, n*)* with an additional parameter *lcp*, reverses the *lcp* array in the same order as A is reversed. The array *state* is initialized to "false".

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.

The algorithm *Reverse* in Fig. 3 reverses a suffix array as follows: the value of i is written in the entry $A(i)$ and the old content denotes the next entry where the value $A(i)$ 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 the lcp array, and calculates the LCP values from suffix 1 to suffix n. At line (3), $\mathrm{lcp}(i)$ gives the entry i' of A that contains suffix i. At line (8) $\mathrm{lcp}(i)$ gets the length of the LCP of suffixes $A(i')$ and $A(i'-1)$. After line (9) the LCP values are stored in the suffix number order, and so the array lcp is reversed at line (10) to get the lcp array correspond to the order of suffixes in the suffix array.

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 $i = A(x)$ and $A(x-1)$ in the suffix array A of length n, when $1 < x \leq n$, and let $lcp(i) = 0$, when $i = A(1)$. 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 comparisons to the first mismatching characters (for a more detailed analysis, see [14]).

Proposition 3.2. 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 the suffix array in $O(n)$ time using $n + O(1)$ bits additional work space.

4. Pattern Searches from CSA

During this section, we will assume that a CSA is binary searchable (and omit repeating the term). Then pattern searches from a CSA can be done almost similarly as from suffix arrays. In addition to binary searches for finding the area of the array where the occurrences lie, we also need to be able to uncompact the links to retrieve these occurrences. These operations become non-trivial because of *cyclic links*.

Figure 1 shows a simple example of a cyclic link; suffixes 2 and 3 are replaced by a link to suffixes 1 and 2, 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 (simply because the suffixes grow monotonically in each link path). This is the key property that can be exploited in an algorithm to uncompact links.

Fact 4.1. If an algorithm that uncompacts a link follows only the link paths of the suffixes represented by the link, then the algorithm will not fall into a cycle.

4.1. Binary Search over CSA

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 $l \ldots r$. 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 describe how the links can be uncompacted. The algorithm for the task is given in Fig. 4.

Theorem 4.1. Function *UnCompactLink* works correctly, i.e. it returns an array $\mathrm{UCA}(1 \ldots \Delta)$ that contains the suffixes that are represented by a link $(x, \Delta x, \Delta)$ in a compact suffix array CA.

Proof:
Reminding the Fact 4.1, it is enough to prove that the algorithm follows only the link paths of suffixes represented by the link. 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. Since the root passes parameters *UnCompactLink*(CA,x,0,UCA,1,$\mathrm{CA}(x).\Delta$,0) to its only child, the required information to start uncompacting is available at the beginning; the start of the link path of the first suffix (in variable x) and the number of suffixes to be uncompacted (Δr) are passed. 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 to reach the link path of the desired suffix, i.e. the skip value passed from the parent plus the cumulated skip value in the path from the root to the parent node. Variable Δ gets the value of how many suffixes must be uncompacted in this sub-tree, i.e. 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.

```
UnCompactLink(in CA,in x,in d,in out UCA,in out j,in Δr,in Δp):
    {CA is the compact suffix array, x is the link to be uncompacted,
    d is the current link path length, UCA is where the uncompacted suffixes are to be stored,
    j is the current position in UCA, Δr is number of suffixes remaining to be uncompacted, and
    Δp is the cumulated skip value from the root to the parent.}
(1)  y ← CA(x).x; Δx ← CA(x).Δx + Δp; Δ ← min(CA(x).Δ − Δp, Δr)
(2)  j' ← j; i ← 0
(3)  while Δx > 0 do begin {Skip links and suffixes that are not required.}
(4)      if CA(y + i).B = "suffix" then begin
(5)          i ← i + 1; Δx ← Δx − 1 end
(6)      else begin
(7)          if CA(y + i).Δ ≤ Δx then begin
(8)              Δx ← Δx − CA(y + i).Δ; i ← i + 1 end
(9)          else break end end
(10) while j − j' < Δ do begin {Uncompact until required suffixes are copied.}
(11)     if CA(y + i).B = "link" then begin
(12)         UnCompactLink(CA, y + i, d + 1, UCA, j, Δ − (j − j'), Δx)
(13)         Δx ← 0 end
(14)     else begin
(15)         UCA(j).x ← CA(y + i).x + d + 1; j ← j + 1 end
(16)     i ← i + 1 end
```

Figure 4. An algorithm for uncompacting a link.

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. Search Time Analysis

In the worst case, the searches in the CSA can 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 have introduced a concept of $(\mathcal{C}, \mathcal{D})$ -limited CSA.

Theorem 4.2. The algorithm in Fig. 4 uncompacts a link of a $(\mathcal{C}, \mathcal{D})$-limited CSA using $O(\mathcal{C}^2\mathcal{D})$ time in the worst case.

Proof:
In the worst case, there are $\mathcal{C}$ link paths of length $\mathcal{D}$ starting from a link. Since the algorithm in Fig. 4 follows only these link paths, there are at most $\mathcal{CD}$ links that are traversed. Each of these links may have a skip value $\Delta x = \mathcal{C} - 1$. Thus, there may be at most $(\mathcal{C} - 1)\mathcal{CD}$ suffixes/links that have to be skipped. Since the rest of the computation of the algorithm is proportional to $\mathcal{CD}$, the amount of skipping dominates the time requirement, which gives the result. □

A more constructive analysis reveals that actually $O(\mathcal{C}^2\mathcal{D})$ is a tight bound. An example of a situation where the bound is achieved is shown in Fig. 5.

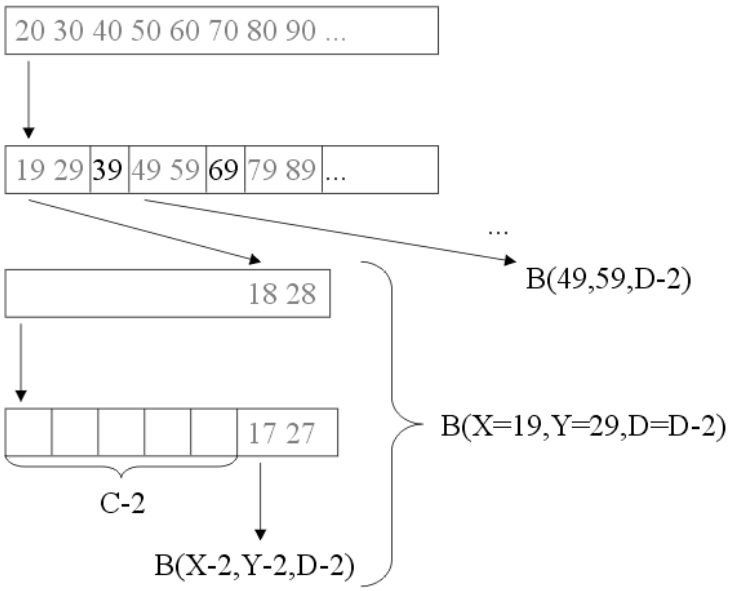

Figure 5. A worst case example for uncompacting a link. On top, there is a link that represents suffixes $20, 30, \ldots$ This link points to an area, where suffixes 19 and 29 are again replaced by a link, 39 is a suffix, and so on. The link that represents suffixes 19 and 29 points inside another link with $\Delta x = \mathcal{C} - 2$. This link then points into an area that consists of $\mathcal{C} - 2$ suffixes (which have to be skipped when uncompacting the link on top) and a link that contains the desired suffixes 17 and 27. The same link structure continues from the link representing 17 and 27. We denote this recursive structure with $B(X, Y, D)$ above, where X and Y are the two suffixes that start the recursion and D is the maximum depth of the recursion. Value D is initialized to $\mathcal{D} - 1$ and will decrease by 2 at each step of the recursion. The recursion will continue until $D < 0$. It is not hard to see that about $(\mathcal{C} - 2)\frac{\mathcal{D}}{2}\frac{\mathcal{C}}{3}$ suffixes must be skipped in this example. Moreover, a text can be constructed such that the above kind of compact suffix array sub-structure does exist.

One can also consider the expected case complexity of uncompacting a link. However, it is not reasonable to make expected case analysis assuming a random text, since a CSA of a random text will take almost the same space as a suffix array. We would like to see the achieved compression as a parameter in the expected case complexity. For this reason, we make some probabilistic assumptions of the nature of the CSA; we assume a *random link distribution*, where the links point into random places in the CSA, and the compression is identically and independently distributed (i.i.d.) over the CSA. These assumptions try to capture the situation where all parts of the suffix array are equally compressible.

Theorem 4.3. The algorithm in Fig. 4 uncompacts a link of a BSCSA of size n' of a text of size n using $O((n/n')^3)$ time in the expected case under the random link distribution.

Proof:
Let X_i denote the length of the linked area number i, and let s be the number of linked areas in the (binary searchable) CSA. We can identify suffixes with links of length 1, and thus $s = n'$. Now, it holds that $\sum_{i=1}^{n'} X_i = n$. Since we assume a random link distribution, we can identify values X_i with i.i.d. random variables and write an equation $\sum_{i=1}^{n'} E(X_i) = n$ for the expected length of the linked areas. We have that $E(X_i) = \frac{n}{n'}$ for each i. Although this result holds for a CSA, it can be easily generalized for a BSCSA; the expected length is now at most $\frac{2n}{n'}$, since the compressed length at most doubles when one suffix is removed from each link to make the CSA binary searchable.

Let Y be a random variable to denote the length of a link path. Under the random link distribution, the probability that a link points to a suffix is at least $\frac{n'}{2n}$, since there are at least $\frac{n'}{2}$ suffixes out of n present in the BSCSA. Thus, the probability of a link path of length k is at most $(1 - \frac{n'}{2n})^{k-1}\frac{n'}{2n}$. For the expected length, we have an estimate $E(Y) \leq \frac{2n}{n'}$.

From the worst case analysis we know that the amount of work needed to uncompact a link is bounded by the amount of skipping. The amount of skipping needed for uncompacting link i is bounded by

$$W(i) = \sum_{j=1}^{X_i} \sum_{k=1}^{Y_{i,j}} X_{i_{j,k}}, \tag{1}$$

where X_i denotes the length of linked area number i, each $Y_{i,j}$ denotes the length of the link path of the j^{th} suffix of link i, and each $X_{i_{j,k}}$ denotes the length of the k^{th} linked area at the link path of the j^{th} suffix of link i (i.e. an upper bound for the skip value). Now, we can again identify these values with random variables to get an equation for the expected amount of work. These random variables are not independent, which makes it difficult to solve the expected value $E(W(i))$ exactly. The crucial observation is that the dependency can only make $E(W(i))$ smaller; same link paths, and therefore also the amount of work to skip to the start of a link path, can be shared by consecutive suffixes. Thus, we can replace X_i, each $Y_{i,j}$, and each $X_{i_{j,k}}$ with independent random variables X, Y, and Z, respectively, where X and Z each denote the length of a linked area, and Y denotes the length of a link path. We have en estimate

$$E(W(i)) \leq E(XYZ) = E(X)E(Y)E(Z) \leq (\frac{2n}{n'})^3. \tag{2}$$

□

Now, we can summarize the required search times.

Corollary 4.1. To report $\mathrm{occ} > 0$ occurrences of a pattern of length m in a text of length n using a $(\mathcal{C}, \mathcal{D})$-limited CSA of size n' takes $O(m \log C + \log n' + \mathcal{C}^2\mathcal{D} + \mathrm{occ}\,\mathcal{C}\mathcal{D})$ time in the worst case. To get the time needed for *existence queries* (to determine whether $\mathrm{occ} > 0$), set $\mathrm{occ} = 0$.

Proof:
When the LCP values are computed for the CSA, the first two binary searches take $O(m + \log n')$ time using an identical algorithm to the one given in [18]. After that, we have an area $l \ldots r$ where all links must be uncompacted. Consider first the existence query. If the area is empty, the pattern does not occur

in the text. If the area is longer than one, we have found a match; the structure of the CSA guarantees that there is at least one suffix in this area, and hence the binary search could have continued. Thus, in the worst case we have one link to uncompact; it takes $O(\mathcal{C}^2\mathcal{D})$ time. After the link is uncompacted, we still need to make a binary search over the uncompacted suffixes to search for the pattern. This takes $O(m \log \mathcal{C})$ time since we do not have the LCP values for these suffixes. Summing all factors together, we get the desired bound for existence queries.

For reporting the occurrences, we need to uncompact all links between the left and right limits and search for the start and end points of the occurrences. If the left and right limits contain suffixes, then we have already found the start and end points. In the case that the left (resp. right) limit is a link, then the start (end) of the occurrences must be inside that link. Otherwise the first (second) binary search could have continued. In the worst case, we only need to uncompact two links and make one binary search on each uncompacted area to find the start and end points of the occurrences. This adds $O(\mathcal{C}^2\mathcal{D} + m \log \mathcal{C})$ to the search time. The time needed for uncompacting the remaining links can be partly charged on the occurrences; it takes $O(\mathrm{occ}\, \mathcal{C}\mathcal{D})$ time, since $\mathrm{occ} \geq l \times \mathcal{C}$, where l is the amount of links that contain occurrences. We have now the desired bound for reporting the occurrences. □

Using the same arguments as in the above proof we obtain:

Corollary 4.2. To report $\mathrm{occ} > 0$ occurrences of a pattern of length m in a text of length n using a CSA of size n' takes $O(m \log \frac{n}{n'} + \log n' + (\frac{n}{n'})^3 + \mathrm{occ}(\frac{n}{n'})^2)$ time in the expected case, assuming a random link distribution. To get the time needed for existence queries, set $\mathrm{occ} = 0$.

To ease the comparison to other index structures, we can choose $\mathcal{D} = \lceil \log \lceil \log n \rceil \rceil$ and $\mathcal{C} = 2^{\mathcal{D}} - 1$. Now, it takes $O(m \log \log n + (\log n)^2 \log \log n + \mathrm{occ} \log n \log \log n)$ time to report $\mathrm{occ} > 0$ occurrences. If the text is compressible, the index can take $O(\frac{n}{\mathcal{C}} \log \frac{n}{\mathcal{C}}) = O(n)$ bits.

5. Implementation

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 $\mathcal{C}$ and $\mathcal{D}$ to limit the lengths of linked areas and link paths. It follows that values Δx and Δ can be represented in $\lceil \log(\mathcal{C}+1) \rceil$ 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 $n' \lceil \log(\mathcal{C}+1) \rceil$ (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 $\mathcal{C} \in \{2^0 - 1, 2^1 - 1, \ldots, 2^{\lceil \log n \rceil} - 1\}$ ($\mathcal{D} = \infty$) and choose the value that results into the minimal CSA. Therefore we can construct the minimal CSA in $O(n \log n)$ time.[2] A minimal CSA takes

[2]One might argue that $O(n \log \log n)$ could suffice for constructing a minimal CSA by using binary search over the possible values for $\mathcal{C}$. However, the size of CSA is not a simple function that first decreases and then increases, as one would easily believe. It is possible, but tedious, to construct an example where the function has multiple local minima.

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 Space Requirement of Construction

A straightforward implementation of the construction algorithm in Sect. 3.1 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 a $(\mathcal{C}, \mathcal{D})$ -limited CSA, we need an additional array for limiting the lengths of the link paths. Because the use of the value $\mathcal{D}$ is only worthwhile when the value is small, we can assume that $\mathcal{D}$ fits into a byte. Therefore the whole construction uses $9n$ bytes. When the LCP values are needed, the construction can still be done using $13\frac{1}{8}n$ bytes, by using the algorithms presented in Sect. 3.2 in a proper order; note that the suffix array and the LCP array are not needed at the same time.

6. Experimental Results

We implemented all the methods discussed in this paper. The space-efficient construction algorithms became necessary as we used relatively large datasets. However, the space-efficiency in the structures 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 $\mathcal{C}$ (maximum length of a linked area) and $\mathcal{D}$ (maximum 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 examined the effect of the values $\mathcal{C}$ and $\mathcal{D}$. We ran two tests varying first the values of $\mathcal{C}$ and then $\mathcal{D}$ on text *txtcorpus*. The results are shown in Fig. 6. Instead of measuring time, we calculated the amount of operations that were needed to uncompact the whole array[3]. It can be seen that both parameters affect the uncompacting time considerably, although the behavior is not even near the worst case bound. On the other hand, the expected case analysis is too optimistic, because it is based on the

[3]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.

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 (with line breaks removed from the original file).

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

assumption that all parts of the suffix array are equally compressible, which is not true for suffix arrays of typical 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 the worst case behavior. However, small values for $\mathcal{C}$ and $\mathcal{D}$ force the compression to distribute more evenly, and the expected case bound is then valid.

The compression was not improved significantly as $\mathcal{C}$ and $\mathcal{D}$ grew except when they were small. Usually the minimal CSA is achieved with small $\mathcal{C}$ (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 $\mathcal{D}$.

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.

The LCP values did not provide any significant speed up in our tests. In fact, Table 2 shows that searching was usually slower with the LCP values; the more complicated implementation seems to overrule the effect of the LCP values, especially because accessing the array fields was much more costly than in a standard array implementation. Similar behavior dominated also the searches from the suffix array. To really see the effect of LCP values, we tested also with text $a^{1000000}$ 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 [13] from [9] (with a simple modification to report all the matches). We used the algorithm of Larsson and Sadakane [17] 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 practice. The results are shown in Table 3.

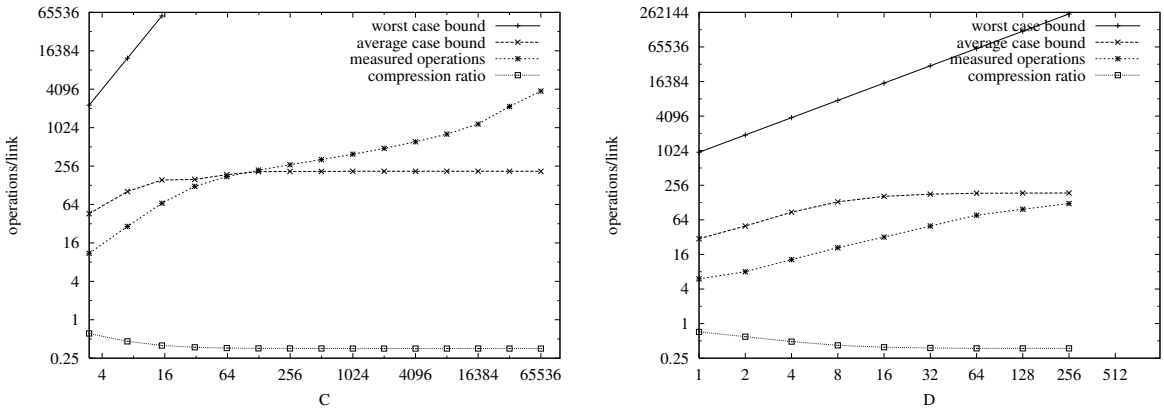

Figure 6. The effect of values $\mathcal{C}$ (left) and $\mathcal{D}$ (right) on uncompacting the CSA of file *txtcorpus*. In the left, the values of $\mathcal{C}$ are $2^2-1, 2^3-1, \ldots, 2^{16}-1$, while $\mathcal{D} = 256$. In the right, the values of $\mathcal{D}$ are $2^1, 2^2, \ldots, 2^8$, while $C = 31$. We measured the amount of skipping and the amount of links that were traversed during the uncompacting. The average value per a link is reported. The worst case bound that corresponds to the measured operations is $\mathcal{C}^2\mathcal{D} + \mathcal{C}\mathcal{D}$. Similarly, the expected case bound is $(\frac{2n}{n'})^3 + (\frac{2n}{n'})^2$. Also the compression ratio $\frac{n'}{n}$ is plotted.

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 gap between sequential search and indexed search is huge, in favor of the index.

7. Discussion and Comparison to Related Work

The large space requirement of full-text indexes is often the reason why they can not 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 the size of the text. When the number of occurrences is small, the searches are only two times slower than from 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 $n\lceil \log n \rceil$ bits as in suffix arrays, making them too expensive in many applications. There are also recent work considering full-text indexes that occupy $o(n \log n)$ bits [25], or $O(n)$ bits [16, 15, 12, 24], or even less [6, 22, 11]. They also have some logarithmic penalties on the search times (except [25], which is theoretically as fast as suffix tree). In addition, they require complicated data structures that may incur large constant factors.

There are implementations available of some of the above indexes, made by the authors of papers [24, 6, 22]: CA-index by Sadakane, FM-index by Ferragina and Manzini, and LZ-index by Navarro. Ferragina and Manzini have made an experimental study of their FM-index [7]. In addition, Navarro [23] has implemented a version of the FM-index, and has made a wide comparison of all the existing implementations. That comparison shows that the CA-index and the FM-index are reasonably fast in existence and counting queries, but really slow in reporting the occurrences. On the other hand, the LZ-index is slow in searching long patterns, but it scales well for reporting the occurrences.

We made a small experiment to see how CSA compares with the other indexes. We

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|	$\mathcal{C}$	$\mathcal{D}$	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	4-10	7	12	79	96	92
07hgp10.txt	30-50	7	12	116	121	116
07hgp10.txt	200-500	7	12	187	176	168

compared CSA with FM-index and with LZ-index. We used the authors' implementations of the indexes, which can be found from *http://butirro.di.unipi.it/∼ferrax/fmindex/* and from *http://www.dcc.uchile.cl/∼gnavarro/software/*, respectively. The default settings were used in all implementations. The CSA of text *world192.txt* plus the actual text took 6.77 times the size of the FM-index, and 2.82 times the size of the LZ-index. These two indexes do not need the original text to function (it is included in compressed form with the index) and for this reason they take significantly less space than CSA. We first tested the existence queries. We searched three patterns — "Finland", "is", and a pattern of length 48 — from text *world192.txt* with the three indexes. First pattern had 78 occurrences, second had 9807, and the third had only one occurrence. The timings for the first pattern were 0.0004ms, 0.03ms, and 0.17ms, for the second pattern they were 0.0004ms, 0.38ms, and 0.03ms, and for the third they were 0.0007ms, 0.3ms, and 1.4ms, with CSA, LZ-index, and FM-index, respectively. Thus, the other indexes were 75–2000 times slower than CSA. Then we tested the reporting queries. Timings for reporting the occurrences of the first pattern were 0.02ms, 0.06ms, and 60ms, for the second pattern they were 2ms, 3ms, and 7901ms, and for the third they were 0.0012ms, 0.4ms, and 2.49ms, with CSA, LZ-index, and FM-index, respectively. The timings were averages over several trials, or big enough numbers for one trial to suffice. This experiment shows the same facts as the experiments in [23]. Namely, that FM-index has a severe bottleneck in reporting the occurrences; it was already 3000 times slower than CSA for reporting 78 occurrences. On the other hand, LZ-index was competetive with CSA in reporting queries except with the long pattern; then LZ-index took 333 times more time.

As a conclusion, CSA is superior to other small indexes in search times. It scales well in both two directions: When pattern size is increased (and thus the number of occurrences is decreased) and when the number of occurrences is increased (which happens usually when pattern size is decreased). This kind of behaviour is not seen in any other small index. A disadvantage is that despite CSA is significantly smaller index than suffix array, it can not compete in space usage with the most succinct ones.

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 $\mathcal{D} = 12$, CSA2 denotes CSA with $\mathcal{D} = \lceil \log \lceil \log n \rceil \rceil$ and $\mathcal{C} = 2^{\mathcal{D}} - 1$, SA denotes the suffix array, and BMH denotes the Boyer-Moore-Horspool search.

	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				5176	67	68	36	46760
paper1	1.29	1.46	2.00	152	52	46	25	1340
paper1				346	51	47	25	1550
pge0112.txt	1.75	1.90	3.00	6173	85	80	48	312580
pge0112.txt				98557	133	116	49	520300
progp	1.01	1.26	2.00	14	54	44	23	1190
progp				34	54	44	24	1370
txtcorpus	1.59	1.84	3.25	10000	106	91	56	1384570
txtcorpus				351354	301	212	60	2647240
world192.txt	1.15	1.41	2.75	1700	131	85	42	130850
world192.txt				35368	150	96	42	148160
07hgp10.txt	2.68	2.87	3.38	10000	115	118	91	4038640
07hgp10.txt				14542	124	124	91	8571530

Acknowledgements

I would like to thank Esko Ukkonen for valuable comments on this work. Also the comments from anonymous reviewers of this and the previous conference versions have helped to improve the work significantly.

References

[1] A. Blumer, J. Blumer, D. Haussler, and R. McConnell. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34:3, pp. 578–595, 1987.

[2] A. Blumer, D. Haussler, and A. Ehrenfeucht. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics*, 24, pp. 37–45, 1989.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.

[4] M. Crochemore. Transducers and repetitions. *Theor. Comp. Sci.*, 45, pp. 63–86, 1986.

[5] M. Crochemore and Renaud Vérin. Direct Construction of Compact Directed Acyclic Word Graphs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997)*, LNCS 1264, pp. 116–129, 1997.

[6] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In Proc. IEEE Symposium on Foundations of Computer Science (FOCS 2000), pp. 390–398, 2000.

[7] P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pp. 269–278, 2001.

[8] 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, pp. 30–42, 1999.

[9] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C*. Addison-Wesley, Wokingham, UK, 1991. (second edition).

[10] 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.

[11] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pp. 841–850, 2003.

[12] 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 (STOC 2000)*, pp. 397–406, 2000.

[13] R. N. Horspool. Practical fast searching in strings. *Soft. Pract. and Exp.*, 10, pp. 501–506, 1980.

[14] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, LNCS 2089, pp. 181–192, 2001.

[15] J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.

[16] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pp. 141–155. Carleton University Press, 1996.

[17] 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.

[18] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.

[19] E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23, pp. 262–272, 1976.

[20] V. Mäkinen. Compact Suffix Array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pp. 305–319, 2000.

[21] V. Mäkinen. Trade Off Between Compression and Search Times in Compact Suffix Array. In *Proc. 3rd Workshop on Algorithm Engineering and Experimentation (ALENEX 2001)*, LNCS 2153, pp. 189–201, 2001.

[22] G. Navarro. Indexing Text using the Ziv-Lempel Trie. In *Proc. 9th Intl. Symp. String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pp. 325–336, 2002.

[23] G. Navarro. The LZ-index: A Text Index Based on the Ziv-Lempel Trie. *Technical Report TR/DCC-2003-1*, Univ. of Chile, Department of Computer Science, January 2003.

[24] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Intl. Symp. Algorithms and Computation (ISAAC'00)*, LNCS 1969, pp. 410–421, 2000.

[25] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, Vol. 82 (6), pp. 307-311, 2002.

[26] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.

[27] P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.