Compressed Compact Suffix Arrays

Veli Mäkinen[1] and Gonzalo Navarro[2*]

[1] Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
`vmakinen@cs.helsinki.fi`
[2] Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile.
`gnavarro@dcc.uchile.cl`

Abstract. The *compact suffix array* (CSA) is a space-efficient full-text index, which is fast in practice to search for patterns in a static text. Compared to other *compressed suffix arrays* (Grossi and Vitter, Sadakane, Ferragina and Manzini), the CSA is significantly larger (2.7 times the text size, as opposed to 0.6–0.8 of compressed suffix arrays). The space of the CSA includes that of the text, which the CSA needs separately available. Compressed suffix arrays, on the other hand, *include* the text, that is, they are *self-indexes.* Although compressed suffix arrays are very fast to determine the *number* of occurrences of a pattern, they are in practice very slow to *report* even a few occurrence positions or text contexts. In this aspect the CSA is much faster. In this paper we contribute to this space-time trade off by introducing the *Compressed CSA* (CCSA), a self-index that improves the space usage of the CSA in exchange for search speed. We show that the occ occurrence positions of a pattern of length m in a text of length n can be reported in $O((m + occ)\log n)$ time using the CCSA, whose representation needs $O(n(1 + H_k \log n))$ bits for any k, H_k being the k-th order empirical entropy of the text. In practice the CCSA takes 1.6 times the text size (and includes the text). This is still larger than current compressed suffix arrays, and similar in size to the LZ-index of Navarro. Search times are by far better than for self-indexes that take less space than the text, and competitive against the LZ-index and versions of compressed suffix arrays tailored to take 1.6 times the text size.

1 Introduction and Related Work

The classical problem in string matching is to determine the occ occurrences of a short pattern $P = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$. Text and pattern are sequences of characters over an alphabet Σ of size σ. In practice one wants to know the text positions of those occ occurrences, and usually also a text context around them. 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.

[*] Supported in part by Fondecyt Grant 1-020831.

To allow fast searches for patterns of any size, the index must allow access to all suffixes of the text. These kind of indexes are called *full-text indexes*. Optimal query time, which is $O(m + occ)$ as every character of P must be examined and the *occ* occurrences must be reported, can be achieved by using the *suffix tree* [19] as the index. In a suffix tree every suffix of the text is represented by a path from the root to a leaf. The space requirement of a suffix tree is very high. It can be $12n$ bytes in practice, even with a careful implementation [7]. In addition, in any practical implementation there is always an alphabet dependent factor on search times.

The *suffix array* (SA) [11] is a reduced form of the suffix tree. It represents only the leaves of the suffix tree, via pointers to the starting positions of all the suffixes. The array is lexicographically sorted by the pointed suffixes. A suffix array takes $4n$ bytes, and searches in $O(m \log n + occ)$ time via two binary searches. One finds the first cell i pointing to a suffix $\geq P$ (lexicographically), and the other finds the first cell j pointing to a suffix $\geq p_1 p_2 \ldots p_{m-1}(p_m + 1)$. Then all the cell values at suffix array positions $i \ldots j - 1$ are the initial positions of occurrences of P in T.

There is often a significant amount of redundancy in a suffix array, such that some array areas can be represented by links to other areas. Basically, it is rather common that one area contains the same pointers of the other area, all shifted by one text position. This observation has been intensively used recently in different ways to obtain succinct representations of suffix arrays and still provide fast search time [8, 18, 5].

The *compact suffix array* (CSA) [14] makes direct use of that redundancy to reduce the space usage of suffix arrays. Areas similar to others (modulo a shift in text positions) are found and replaced by a direct link to the similar areas. In practice the CSA takes less than $2n$ bytes and can search in $O(m \log n + occ)$ time, which in practice turns out to be about twice as slow as the plain suffix array. Note that, like suffix trees and arrays, the CSA needs the text itself separately available.

A recent trend in compressed data structures is that of *self-indexes*, which include the text. Hence the text can be discarded and the index must provide functions to obtain any desired text substring in reasonable time. Self-indexes open the exciting possibility of the index taking less space than the text, even including it. Existing implemented self-indexes are the compressed suffix array CSArray of Sadakane [18] (built on [8]), the FM-index of Ferragina and Manzini [5, 6], and the LZ-index of Navarro [16]. The first two take 0.6–0.8 times the text size, while the LZ-index takes about 1.5 times the text size on English text.

In this paper we introduce the *compressed CSA* (CCSA), a self-index based on the CSA which is more compact and represents a relevant space-time trade off in practice. We retain the links of the CSA, but encode them in a compact form. We also encode the text inside the CCSA by using small additional structures that permit searching and displaying the text without accessing T. We show that the CCSA needs $O(n(1 + H_k \log n))$ bits for any k, and that it can find all the occurrences of P in T in $O((m + occ) \log n)$ time. In an 80 Mb English text example, the CCSA need 1.6n bytes, replacing the text. This is much less than the 2.7n bytes needed by the CSA, about the same space of the LZ-index, and 2–3 times larger than other compressed suffix arrays. Searching the CCSA is 50 times slower than the CSA, but 50–75 times faster than any other self-index that takes less space than the text. The CCSA is competitive against the LZ-index, and against compressed suffix arrays versions tailored to use the same 1.6n space to boost their search time.

Our space analysis represents indeed a contribution with independent interest, as we relate the space requirement of CCSA and CSA to the number of runs in Burrows-Wheeler transformed text [2]. We show that this quantity is at most $|\Sigma|^k + 2H_k n$.

2 The Compact Suffix Array (CSA)

Let Σ be an ordered alphabet of size $\sigma = |\Sigma|$. Then $T = t_1 t_2 \ldots t_n \in \Sigma^*$ is a (text) *string* of length $n = |T|$. A *suffix* of text T is a substring $T_{i \ldots n} = t_i \ldots t_n$. We assume that the last text character is $t_n = \$$, which does not occur elsewhere in T and is lexicographically smaller than any other character in Σ.

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

The idea of compacting the suffix array is the following: Let $\ell \geq 0$. Find two areas $j \ldots j + \ell$ and $i \ldots i + \ell$ of SA that are repetitive in the sense that the suffixes represented by $j \ldots j + \ell$ are obtained, in the same order, from the suffixes represented by $i \ldots i + \ell$ by inserting the first symbol. In other words, $SA[j + k] = SA[i + k] - 1$ for $0 \leq k \leq \ell$. Then replace the area $j \ldots j + \ell$ of SA by a link, stored in $SA[j]$, to the area $i \ldots i + \ell$. This is called a *compacting operation.* The areas may be compacted recursively, meaning that area $i \ldots i + \ell$ (or some parts of it) may also be replaced by a link.

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

- A pointer p to the entry that contains the start of the linked area.
- A value δ such that entry $p + \delta$ denotes the actual starting point after entry p is uncompacted.
- The length of the linked area ℓ.

Definition 2 *A* compact suffix array (CSA) *of text* T *of length* $n = |T|$ *is an array* $CSA[1 \ldots n']$ *of length* $n' \leq n$*, such that for each* entry $1 \leq i \leq n'$, $CSA[i]$ *is either an explicit suffix or a triple* (p, δ, ℓ)*, where* p*,* δ*, and* ℓ *denote a link to an area obtained by a compacting operation from the suffix array of* T*. The* optimal *CSA for* T *is such that its length* n' *is the smallest possible.*

The original idea of using CSA as an index [14] is to guarantee that a CSA is *binary searchable*. That is, not all areas of the suffix array are compacted; it is required that each other entry of the CSA contains a suffix. The search for a pattern then consists of three phases: (i) A binary search is executed over the entries of the CSA that contain suffixes, (ii) the entries in the range found by the initial binary search are uncompacted, and (iii) the start and end of occurrences is found by binary searches over the uncompacted area.

3 The Compressed CSA

The Compressed CSA (CCSA) is conceptually built on top of the CSA. It involves some slight changes in the structure itself, and radical changes in its representation. A complete example is given in Fig. 1

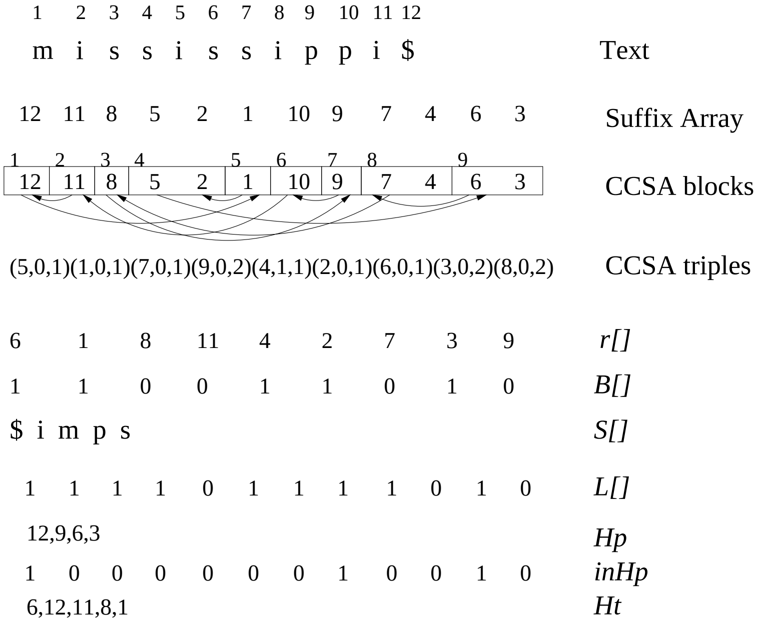

Fig. 1. Example of our CCSA structure for the text `"mississippi$"`.

3.1 Conceptual Structure

The CCSA data structure is conceptually composed of an array of entries (p_i, δ_i, ℓ_i), $1 \leq i \leq n'$, just like the CSA. This array, however, differs slightly from that of the CSA. It corresponds to the *optimal* CSA defined in previous section, without any explicit suffix. The CCSA represents the original suffix array SA as follows. Entry i in the CCSA represents a block of ℓ_i entries of SA, namely entries $\left(\sum_{1\leq j<i}\ell_j\right)+1$ to $\left(\sum_{1\leq j<i}\ell_j\right)+\ell_i$. The actual content is obtained by copying ℓ_i positions of SA from another area and subtracting 1 from their cell values. The pair (p_i,δ_i) is a reference to the SA position where the area to copy begins. The reference indicates position inside the CCSA array and offset inside the p_i-th block (so it should hold $0\leq\delta_i<\ell_{p_i}$). The corresponding absolute SA position is $sapos(p_i,\delta_i)$, where $sapos(p,\delta)=1+\delta+\sum_{1\leq j<p}\ell_j$.

The only case where no proper reference exists is for SA entry with value n. In this case we state that the entry should reference position 1.

Furthermore, the CCSA array has to be of minimum size. That is, it cannot happen that $sapos(p_i,\delta_i)=sapos(p_{i-1},\delta_{i-1})+\ell_{i-1}$, as in this case the CCSA entry i could be merged with entry $i-1$. However, we limit areas that can be extended so that the first characters of all the suffixes pointed by the SA area represented by a single CCSA entry are equal.

Conceptually, the CCSA structure needs the text separately available. However, we propose now a representation both to compress the CCSA and to get rid of the explicit representation of T.

3.2 A Compact Representation

The CCSA array will be represented as follows. For each block (p_i,δ_i,ℓ_i) we will store number $r_i=sapos(p_i,\delta_i)$, which gives the absolute SA position where the i-th CCSA block points to. Additionally, an array L of n bits will signal the SA positions that start a block in the CCSA. That is, $L[j]=1$ iff there is a value $1\leq i\leq n'$ such that $sapos(i,0)=j$ in the CCSA.

We will be interested in performing *rank* and *select* queries over array L. These are defined as follows: $rank(L,j)$ is the number of 1's in L up to position j, and $select(L,i)$ is the position j of the i-th "1" in L. It is possible to preprocess L so that, using only $o(n)$ additional space, *rank* and *select* queries can be answered in constant time [13, 3].

Now, the components of triple (p_i,δ_i,ℓ_i) can be computed as follows. First, $p_i=rank(L,r_i)$, that is, the number of blocks beginnings up to position r_i in the SA. Second, $\delta_i=r_i-select(L,p_i)$, since $select(L,p_i)$ gives the initial position of the block where r_i points inside. Finally, $\ell_i=select(L,i+1)-select(L,i)$, which is the distance from the current block beginning to the next.

In order to discard the text, we need to supply a structure to replace it. It turns out that we will never need to access T_j directly but, rather, given suffix array entry $SA[i]$, we will access $T_{SA[i]}$. This is much easier, because the characters of T are sorted by index i, that is, given two text characters $a<b$, all the text occurrences of a appear before those of b in the SA. Moreover, since the first characters of each CCSA block are the same, we will only require the characters of the form $T_{SA[sapos(i,0)]}$.

We store an array B of n' bits, so that $B[i]=1$ iff $T_{SA[sapos(i,0)]}\neq T_{SA[sapos(i-1,0)]}$ or $i=1$, that is, if the first character of suffixes in CCSA block i differ from that in the previous block. We also store an array of characters S, of size at most σ, where all the distinct characters appearing in T are stored in lexicographic order. Hence, $T_{SA[sapos(i,0)]} = S[rank(B,i)]$, since $rank(B,i)$ tells how many times the first suffix character has changed since the beginning of the CCSA array, and S maps this number to the corresponding character. Therefore, bit array B will be also preprocessed for $rank$ queries.

The above structures require $n' \log n + n + n' + \sigma \log \sigma + o(n)$ bits. [1] With them we have enough information to determine the SA range that contains the occurrences of a pattern P. In the following we will depict the search algorithms. Later, we will consider the problem of showing the text positions and contexts for the occurrences, and introduce a few more structures for that.

3.3 Search Algorithm

Our aim is to binary search the CCSA just like the SA. Even if the SA is not explicitly represented, we can perform such a binary search provided we are able to extract the first m characters of a given entry $SA[i]$, so as to compare it against our search pattern P. Therefore, our problem is to extract $T_{SA[i]\ldots SA[i]+m-1}$ without having T nor SA.

Let us first concentrate in obtaining character $T_{SA[i]}$. Let $j = rank(L,i)$ be the CCSA block that contains SA entry i. The offset corresponding to entry i inside CCSA block j is $\delta = i - select(L,j)$, so $i = sapos(j,\delta)$. Since all the first letters of blocks inside CCSA block j are the same, we can rather fetch character $T_{SA[sapos(j,0)]}$. As explained above, this is precisely $S[rank(B,j)]$. Hence we can obtain the first character $T_{SA[i]} = S[rank(B,j)]$.

We need now to move to the next character $T_{SA[i]+1}$. But this is easy to to obtain from the CCSA. Since $SA[i]$ corresponds to reference (j,δ) in the CCSA, then position $SA[i]+1$ corresponds to CCSA reference $(p_j, \delta_j + \delta)$. The corresponding SA entry is thus $sapos(p_j, \delta_j + \delta) = r_j + \delta$.

Hence, the algorithm obtains the m characters by repeatedly computing $j \leftarrow rank(L,i)$, getting character $S[rank(B,j)]$, and then moving to $i \leftarrow r_j + i - select(L,j)$. This clearly takes $O(m)$ time, and the whole binary search takes $O(m \log n)$.

3.4 Reporting Occurrence Positions

Once we determine the SA range where the occurrences of P lie, we wish to show those text positions where P occurs. With the current structures we do not have enough information to do that.

We sample text positions at regular intervals of length I, that is, text positions $h + I$, $h + 2I$, ..., so that text position n is sampled, $h = n \bmod I$. For each sampled text position pos, pointed to by SA entry i, we store (i, pos) in an array H_p, in increasing i order. At reporting time, given a position i of SA to report, we search for i in H_p. If present, we immediately know its text position pos. Otherwise, we switch to $i' \leftarrow r_j + i - select(L,j)$, where $j = rank(L,i)$, which is the SA position pointing to text position $pos + 1$ (we do not yet know

[1] Our logarithms are all in base 2.

pos), and repeat the process. If we find (i', pos') in H_p, then the original text position is $pos' - 1$. We repeat the process until we find a reference in array H_p.

Fast searching of array H_p is possible by storing a bit array $inH_p[1 \ldots n]$, such that $inH_p[i] = 1$ iff entry (i, pos) is present in H_p. If present, it is at H_p entry number $rank(inH_p, i)$, since H_p entries are stored in increasing order of i. Hence inH_p is precomputed to answer $rank$ queries in constant time. We note that only pos has to be stored in H_p, since i is actually the search key.

If we sample one text position out of $I = \log n$, then we can execute at most $\log n$ steps in our quest for the text position, since some text position must be sampled in the range $pos \ldots pos + \log n - 1$. Hence the total cost of the process is $O(\log n)$. The extra space needed is $2n + o(n)$ bits, since each of the $n/\log n$ text positions needs $\log n$ bits for pos and inH_p needs $n + o(n)$ bits.

3.5 Showing Text Contexts

Since the CCSA is a self-index, we must be able to show not only the text context around an occurrence, but any text substring we are asked to. Say that, in general, we wish to show a text string of length ℓ starting at text position pos, that is, retrieve $T_{pos \ldots pos+\ell-1}$.

When we considered the binary search, we saw that we can retrieve as many characters as we wish from the suffix pointed to by $SA[i]$, given i. This time, however, we are given $pos = SA[i]$ instead of i, so the first step is to find some suitable i.

We store in array H_t the same entries (i, pos) implicitly stored in H_p, this time in increasing order of pos. Actually, pos does not need to be stored since at array position j we have $pos = h + jI$. Hence, at position $H_t[\lfloor (pos - h)/I \rfloor]$ we find entry (i, pos'), where pos' is the largest sampled text position $pos' \leq pos$. (For this to work properly we must add an entry $H_t[0]$ corresponding to text position 1.) Then, we can extract $\ell + pos - pos'$ text characters from $SA[i]$ with the same method used in the binary search. This will give us $T_{pos \ldots pos+\ell-1}$ as desired. The overall time is $O(\ell + \log n)$ and we need other n bits to store the entries of H_t.

3.6 The Whole Picture

Our final CCSA structure is composed of the following elements:

- Array r of n' entries r_i.
- Array L of n bits with structures for $rank$ and $select$ operations.
- Array B of n' bits with structures for $rank$ operations.
- Array S of at most σ characters.
- Array H_p storing $1 + \lfloor n/\log n \rfloor$ values i, plus bit vector inH_p of n bits with structures for $rank$ operation.
- Array H_t, storing $1 + \lfloor n/\log n \rfloor$ values pos.

Together, these structures add $n' \log n + 4n + n' + \sigma \log \sigma + o(n)$ bits. We remark that the text needs not be stored separately. It is clear that the CCSA can be built in $O(n)$ time from the suffix array, since the most complex part is similar to the CSA construction, which can be done in linear time [14].

We can do better in terms of space, at least in theory. A bit array of size n where only k bits are set can be preprocessed for constant-time *rank* and *select* queries and stored in $\log \binom{n}{k} + o(n)$ bits [1]. In particular, our array B requires only $O(\sigma \log n')$ space, while array inH_p requires $O(n \log \log n / \log n) = o(n)$ space.

The final result, taking σ as a small constant to simplify, is that we need $n' \log n + 3n + o(n)$ bits. With this CCSA structure, we can search for the *occ* occurrences of a pattern of length m and show a text context of length ℓ around each occurrence in worst-case time $O((m \log n + occ(\ell + \log n)))$. If we only want to show the text positions, the complexity is $O((m + occ) \log n)$. If we only want to know how many occurrences there are, the complexity is $O(m \log n)$.

We can attain $n' \log n + n + o(n)$ space by sampling one out of $\log n \log \log n$ entries in arrays H_p and H_t. In this case the time to report the occurrences raises to $O(occ \log n \log \log n)$, and a text string can be displayed in $O(\ell + \log n \log \log n)$ time.

All our space analysis is given in terms of n'. In the next section we show that $n' = O(H_k n)$, and therefore the CCSA structure needs $O(n(1 + H_k \log n))$ bits of space.

4 An Entropy Bound on the Length of CSA and CCSA

We will now prove that the length n' of the optimal CSA and the CCSA is at most $|\Sigma|^k + 2H_k n$, where H_k is the *k-th order empirical entropy* of T [12]. To be precise, we obtain the bound when the indexes are built on the *inverse string* $T^{-1} = t_1^{-1} t_2^{-1} \cdots t_n^{-1} = t_n t_{n-1} \cdots t_1$ of T.

Let us first recall some basic facts and definitions from [12]. Let n_i denote the number of occurrences in T of the i-th symbol of Σ. The zero-order empirical entropy of the string T is

$$H_0(T) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log \frac{n_i}{n}, \tag{1}$$

where $0 \log 0 = 0$. If we use a fixed codeword for each symbol in the alphabet, then $H_0 n$ bits is the smallest encoding one can achieve for T ($H_0 = H_0(T)$). If the codeword is not fixed, but it depends on the k previous symbols that may precede it in T, then $H_k n$ bits is the smallest encoding one can achieve for T, where $H_k = H_k(T)$ is the k-th order empirical entropy of T. It is defined as

$$H_k(T) = \frac{1}{n} \sum_{W \in \Sigma^k} |W_T| H_0(W_T), \tag{2}$$

where W_T is a concatenation of all symbols t_j (in arbitrary order) such that Wt_j is a substring of T. String W is the *k-context* of each such t_j. Note that the order in which the symbols t_j are permuted in W_T does not affect $H_0(W_T)$, and hence we have not fixed any particular order for W_T.

The *Burrows-Wheeler transform* [2], denoted by $bwt(T)$, is a permutation of the text. Run-length encoding of $bwt(T)$ is closely related to the compression achieved by the CSA. The *runs* in $bwt(T)$ (maximal repeats of one symbol) correspond to links in the CCSA; if we construct the optimal CCSA for string T with the restriction that the suffixes inside each linked area must start with the same symbol, then the length of the CCSA is equal to the number of runs in $bwt(T)$. To state this connection formally, recall from [12] that $bwt(T) = t^{-1}_{SA[1]-1}t^{-1}_{SA[2]-1}\cdots t^{-1}_{SA[n]-1}$, where $t^{-1}_0 = t^{-1}_n = \#$ and SA is the suffix array of T^{-1}. Symbol $\# \notin \Sigma$ precedes all symbols of Σ in the lexicographic order. [2] Now, if suffixes $SA[j], SA[j+1], \ldots, SA[j+\ell]$ are replaced by a link to suffixes $SA[i], SA[i+1], \ldots, SA[i+\ell]$ in CCSA, then $SA[j+r] = SA[i+r]-1$ and $t^{-1}_{SA[i+r]-1} = t^{-1}_{SA[i+r']-1}$ for all $0 \leq r, r' \leq \ell$. Since the linked areas are maximal in CCSA, each run in $bwt(T)$ corresponds to exactly one link in CCSA (omitting the degenerate case of t_n). Thus, the length n' of the optimal CCSA equals the number of runs in $bwt(T)$.

We will now prove that the number of runs in $bwt(T)$ is at most $|\Sigma|^k + 2H_k n$.

Let $rle(S)$ be the *run-length encoding* of string S, that is, a sequence of pairs (s_i, ℓ_i) such that $s_i s_{i+1} \cdots s_{i+\ell-1}$ is a *maximal run* of symbol s_i (i.e., $s_{i-1} \neq s_i$ and $s_{i+\ell} \neq s_i$), and all such maximal runs are listed in $rle(S)$ in the order they appear in S. The length $|rle(S)|$ of $rle(S)$ is the number of pairs in it. Notice that $|rle(S)| \leq |rle(S_1)| + |rle(S_2)| + \cdots + |rle(S_p)|$, where $S_1 S_2 \cdots S_p = S$ is any partition of S.

Recall string W_T as defined in Eq. (2) for a k-context W of a string T. Note that we can apply any permutation to W_T so that (2) still holds. Now, $bwt(T)$ can be given as a concatenation of strings W_T for $W \in \Sigma^k$, if we fix the permutation of each W_T and the relative order of all strings W_T appropriately [12]. As a consequence, we have that

$$|rle(bwt(T))| \leq \sum_{W \in \Sigma^k} |rle(W_T)|, \tag{3}$$

where the permutation of each W_T is now fixed by $bwt(T)$. In fact, Eq. (3) holds also if we fix the permutation of each W_T so that $|rle(W_T)|$ is maximized. This observation gives a tool to upper bound $|rle(bwt(T))|$ by the sum of code lengths when zero-order entropy encoding is applied to each W_T separately. We next show that $|rle(W_T)| \leq 1 + 2|W_T|H_0(W_T)$.

First notice that if $|\Sigma_{W_T}| = 1$ then $|rle(W_T)| = 1$ and $|W_T|H_0(W_T) = 0$, so our claim holds. Let us then assume that $|\Sigma_{W_T}| = 2$. Let x and y ($x \leq y$) be the number of occurrences of the two letters, say a and b, in W_T, respectively. We

[2] We follow the convention of Manzini [12]; the original transformation [2] uses T instead of T^{-1}.

have that

$$H_0(W_T) = -(x/(x+y))\log(x/(x+y)) - (y/(x+y))\log(y/(x+y)) \geq x/(x+y), \quad (4)$$

since $-\log(x/(x+y)) \geq 1$ (because $x/(x+y) \leq 1/2$) and $-(y/(x+y))\log(y/(x+y)) > 0$. The permutation of W_T that maximizes $|rle(W_T)|$ is such that there is no run of symbol a longer than 1. This makes the number of runs in $rle(W_T)$ to be $2x+1$. By using Eq. (4) we have that

$$|rle(W_T)| \leq 2x + 1 = 1 + 2|W_T|x/(x+y) \leq 1 + 2|W_T|H_0(W_T). \quad (5)$$

We are left with the case $|\Sigma_{W_T}| > 2$. This case splits into two sub-cases: (i) the most frequent symbol occurs at least $|W_T|/2$ times in W_T; (ii) all symbols occur less than $|W_T|/2$ times in W_T. Case (i) becomes analogous to case $|\Sigma_{W_T}| = 2$ once x is redefined as the sum of occurrences of symbols other than the most frequent. In case (ii) $|rle(W_T)|$ can be $|W_T|$. On the other hand, $|W_T|H_0(W_T)$ must also be at least $|W_T|$, since it holds that $-\log(x/|W_T|) \geq 1$ for $x \leq |W_T|/2$, where x is the number of occurrences of any symbol in W_T. Therefore we can conclude that Eq. (5) holds for any W_T.

Combining Eqs. (2) and (5) we get the following result:

Theorem 3 *The length of the run-length encoded Burrows-Wheeler transformed text of length n is at most $|\Sigma|^k + 2H_k n$, for any fixed $k \geq 1$.*

As a direct consequence of Theorem 3

$$n' \leq |rle(bwt(T))| \leq |\Sigma|^k + 2H_k n, \quad (6)$$

where n' is the length of the optimal CCSA (or CSA) for text T^{-1}.

5 Implementation and Experiments

We implemented our CCSA structure almost exactly as described. The main difference is that we changed the constant time *select* implementation described in [13, 3], as it has a huge constant factor (an asymptotic constant that is usually > 300). Instead, we implemented a tailored algorithm to compute $i - select(L, rank(L, i))$, which is the way we use *select*. In this case we know position i and simply want the last bit set before position i in array L. We implemented a word-wise followed bit-wise upward scan from position i until the first bit set appears. Currently we have only implemented the counting of occurrences and reporting of text positions, but not yet displaying the context around the occurrences. The implementation is available at `http://www.cs.helsinki.fi/u/vmakinen/software/`.

We tried out several alternative implementations for reporting the occurrences. The main idea in these alternative implementations is to exploit the common search paths for consecutive suffixes. This property is used in the original recursive reporting algorithm for compact suffix arrays [14]. We implemented an analogous recursive reporting algorithm for CCSA, but it was only slightly faster than the direct method described in Sect. 3.4. However, an algorithm that only exploits the common search paths for minimizing the (costly) computation of $i - select(L, rank(L, i))$ turned out to be practical; it is about 25% faster than the direct computation.

Our experiments were run over 83.37 Mb of text obtained from the "ZIFF-2" disk of the TREC-3 collection [9]. The tests ran on a Pentium IV processor at 2 GHz, 512 Mb of RAM and 512 Kb cache, running Linux SuSE 7.3. We compiled the code with `gcc 2.95.3` using optimization option `-O3`. Times were averaged over 10,000 search patterns. As we work only in main memory, we only consider CPU times. The search patterns were obtained by pruning text lines to their first m characters. We avoided lines containing tags and non-visible characters such as '&'.

The CCSA index takes 1.6 times the text size. Some quick tests showed that the CCSA is about 50 times slower than the CSA (2.7 times the text size) and 50 to 75 times faster than the standard implementations of the FM-index [5, 6] and the CSArray [18] using default parameters (around 0.7 times the text size). This shows that the CCSA is a valid trade off alternative.

A much more interesting experiment is to determine *how well* does the CCSA use the space it takes. Both the FM-index and the CSArray can be tuned to use more space, so the natural question is how would the CCSA compare against them if we let them use $1.6n$ bytes. Similarly, the LZ-index takes $1.5n$ bytes over our text, so a direct comparison is fair.

The original FM-index implementation (`http://butirro.di.unipi.it/~ferrax/fmindex/`) does not permit using as much as $1.6n$ bytes. Instead, we used the implementation from G. Navarro (`http://www.dcc.uchile.cl/~gnavarro/software`), which takes more space than the text and makes good use of it (see details in [17]), and tuned it to use $1.5n$ bytes. On the other hand, the CSArray original implementation by K. Sadakane (also available at `http://www.dcc.uchile.cl/~gnavarro/software`), let us tuning it to use near $1.6n$ bytes.

Figure 2 shows the result for counting queries (just telling the number of occurrences) and for reporting queries (telling also all the text positions where they appear). For counting, the CCSA is much faster than the LZ-index, albeit slower than the FM-index and the CSArray. It is interesting that the search cost of the CCSA seems to grow slower with m: For $m = 5$ it is 5–15 times slower, but for $m = 60$ it is only 1.5–4 times slower. The reason is evidently in the expected running time; for larger m, only small portion of the pattern is compared against each suffix in the binary search.

For reporting, the CCSA is about 3.5 faster than the FM-index to process each occurrence. This is clear for $m = 5$, where the number of occurrences is high and reporting them dominates overall time. For $m > 20$ their number is low enough to make the counting superiority of the FM-index to show up and dominate the CCSA. The situation is reversed with the LZ-index, which is

10 times faster than the CCSA at reporting occurrences, but its inferiority to find them shows up for $m > 10$, where it loses against the CCSA. Finally, the CSArray is consistently nearly twice as fast as the CCSA.

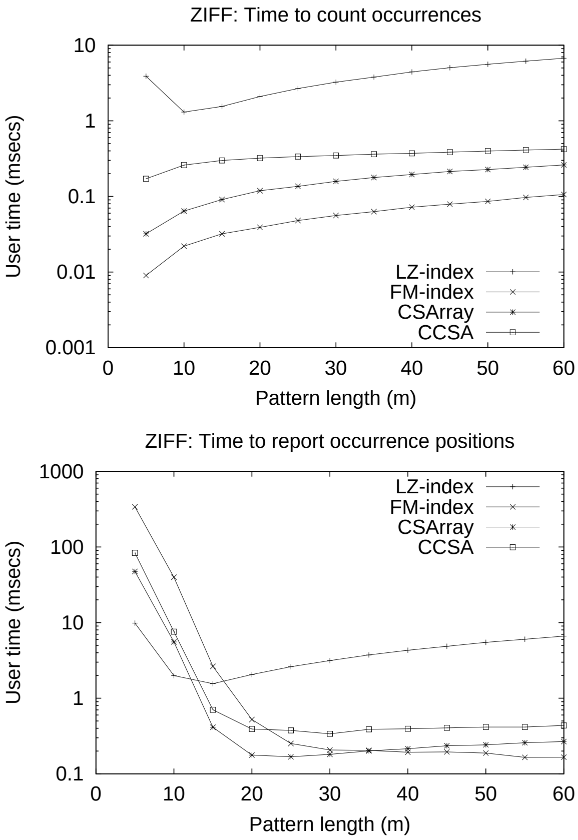

Fig. 2. Query times for our CCSA versus alternative succinct indexes tuned to use about the same space.

6 Conclusions

Compact suffix array represents an analogous improvement to suffix arrays as compact DAWG [4] for suffix trees; both are examples of *concrete optimization* (using the terminology of Jacobson [10]). The research on compressed index structures has recently concentrated on compressing suffix arrays and trees. Such compression is called *abstract optimization* ([10]), as an analogy to the goal to represent a data structure in as small space as possible while supporting the functionality of the abstract definition of the structure.

In this paper, we have presented the first data structure, compressed compact suffix array, that simultaneously exploits *both* concrete optimization and abstract optimization. The resulting structure is competitive against the counterparts that only use abstract optimization.

Our experiments, however, reveal that the structure does not in practice dominate the best current implementations on any domain. Namely, the compressed suffix array implementation of Sadakane [18] is always slightly better. We note that the situation might easily change: Our structure uses heavily the select-function. A more efficient implementation of this function would make our structure a good alternative. Also, if the link structure could be compressed to $O(H_k n)$ bits instead of the $O(H_k n \log n)$ bits, our structure would become very appealing.

The entropy bound on the size of compact suffix array is itself interesting. It could be possible to obtain similar bound also for the size of compact DAWGs, to explain the well-known fact that compact DAWGs have usually much less nodes than suffix trees.

In our subsequent work [15], we have developed an index that is a cross between CCSA and FM-index [5,6]. From the same entropy analysis as used here follows that this index occupies $O(n + H_k n \log |\Sigma|)$ bits. It supports counting queries in time $O(m \log |\Sigma|)$, and reports *occ* occurrences in time $O(occ \log |\Sigma| \log n)$.

References

1. A. Brodnik and I. Munro. Membership in constant time and almost-minimum space. *SIAM J. on Comp.* 5:1627–1640, 1999.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.
3. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.
4. M. Crochemore and Renaud Vérin. Direct Construction of Compact Directed Acyclic Word Graphs. In *Proc. CPM'97*, Springer-Verlag LNCS 1264, pp. 116-129, 1997.
5. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS'00)*, pp. 390–398, 2000.
6. P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *Proc. 12th Symposium on Discrete Algorithms (SODA'01)*, pp. 269–278, 2001.
7. R. Giegerich, S. Kurtz, and J. Stoye. Efficient Implementation of Lazy Suffix Trees. In *Proc. 3rd Workshop on Algorithmic Engineering (WAE'99)*, LNCS 1668, pp. 30–42, 1999.
8. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Symposium on Theory of Computing (STOC'00)*, pp. 397–406, 2000.
9. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. TREC-3*, pages 1–19, 1995. NIST Special Publication 500-207.

10. G. Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.
11. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, pp. 935–948, 1993.
12. G. Manzini. An Analysis of the Burrows-Wheeler Transform. *J. of the ACM* 48(3):407–430, 2001.
13. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
14. V. Mäkinen. Compact Suffix Array — A Space-efficient Full-text Index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.
15. V. Mäkinen and G. Navarro. New search algorithms and time/space trade offs for succinct suffix arrays. *Technical report C-2004-20*, Dept. of Computer Science, Univ. of Helsinki, April 2004.
16. G. Navarro. Indexing Text using the Ziv-Lempel Trie. In *Proc. 9th String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pp. 325–336, 2002. Extended version to appear in *J. of Discrete Algorithms.*
17. G. Navarro. The LZ-index: A Text Index Based on the Ziv-Lempel Trie. *Technical Report TR/DCC-2003-1*, Dept. of Computer Science, Univ. of Chile, January 2003.
18. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Algorithms and Computation (ISAAC'00)*, LNCS 1969, pp. 410–421, 2000.
19. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.