Compact Suffix Array*

Veli Mäkinen

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

Abstract. *Suffix array* is a data structure that can be used to index a large text file so that queries of its content can be answered quickly. Basically a suffix array is an array of all suffixes of the text in the lexicographic order. Whether or not a word occurs in the text can be answered in logarithmic time by binary search over the suffix array. In this work we present a method to compress a suffix array such that the search time remains logarithmic. Our experiments show that in some cases a suffix array can be compressed by our method such that the total space requirement is about half of the original.

1 Introduction

The string-matching problem considered in this paper is to determine for a pattern P and a text T, whether there is a substring inside text T that matches pattern P. If the same text is queried several times, it is worthwhile to build a data structure (index) for the text so that queries can be answered quickly. Optimal inquiry time is $O(|P|)$, as every character of P must be checked. This can be achieved by using the *suffix tree* [13, 10, 11] as the index. In a suffix tree every suffix of the text is presented by a path from the root to a leaf.

The suffix array [9, 8] 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. Although it saves space, the query times get worse. The tree traversal is simulated by a binary search; a branch in a suffix tree can be selected in constant time (or at least in time proportional to the size of the alphabet), but in a suffix array a branch must be searched by a binary search. The search time with the suffix array is therefore $O(|P| \log |T|)$. Manber and Myers [9] have shown that by saving additional information about consecutive suffixes of the binary search in the suffix array, it is possible to search pattern P in time $O(|P| + \log |T|)$.

A suffix tree contains a lot of repetitive information in different tree branches. The uncompacted version of the suffix tree called *suffix trie* can be transformed into a structure called *suffix automaton* or *DAWG (Directed Acyclic Word Graph)* [2] by combining its similar subtrees. If the method is applied straight to the suffix tree we get a *Compact DAWG* (*CDAWG*) [3–6]. This kind of combining can also be applied to the suffix array. As the suffix array represents only the leaves of the suffix tree, there are less possibilities for combining.

* A work supported by the Academy of Finland under grants 44449 and 22584.

In this paper we present a modification 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 feature of the CSA is that the compression does not change the asymptotic search times; searching pattern P from the CSA takes still time $O(|P| + \log |T|)$. Moreover, there is a trade off between the compression performance and the search time in our algorithms. The trade off is controlled using certain constant parameters.

In Section 2 we define the CSA. In Section 3 we give an algorithm to construct a CSA in linear time from the suffix array. In Section 4 we give a general algorithm to search pattern P from the CSA in average time $O(|P| + \log |T|)$. In Section 5 we consider the space requirement and the trade off between compression and search times, and give a practical search algorithm that works in $O(|P| + \log |T|)$ time in the worst case. Experimental results are given in Section 6 both about the size of the structure and the search times. Conclusions are given in Section 7.

2 Compact Suffix Array

Let Σ be an ordered alphabet and $T = t_1 t_2 \ldots t_n \in \Sigma^*$ a text of length $n = |T|$. A *suffix* of text T is a substring $T_{i \ldots n} = t_i \ldots t_n$. A suffix can be identified by its *starting position* $i \in [1 \ldots n]$. A *prefix* of text T is a substring $T_{1 \ldots i}$. A prefix can be identified by its *end position* $i \in [1 \ldots n]$.

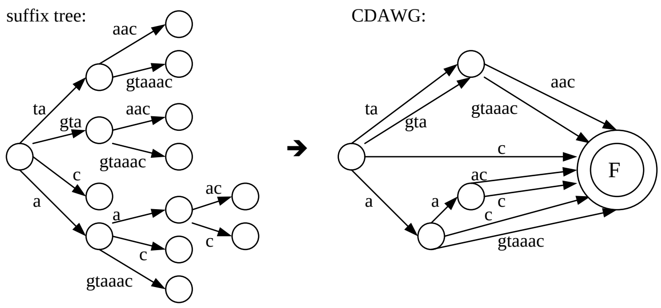
Definition 1 *The* suffix array *of text T of length $n = |T|$ is an array $A(1 \ldots n)$, such that it contains all starting positions of the suffixes of the text T such that $T_{A(1) \ldots n} < T_{A(2) \ldots n} < \ldots < T_{A(n) \ldots n}$, i.e. the array A gives the lexicographic order of all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

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*. Similarly define *uncompacting operation* such that a link to the area $b \ldots b + \Delta$ is replaced by suffixes $a \ldots a + \Delta$.

Basically a compacting operation on a suffix array corresponds to the operation of combining subtrees of a suffix tree to form a CDAWG. This is visualized in Fig. 1(a). Figure 1(b) shows the same operation with a suffix array. As can be seen in Fig. 1, conversions from a suffix tree to a CDAWG, and from a suffix array to a CSA, are slightly different.

In Fig. 1(b), the link at entry 7 of the compact suffix array is to the area that itself is represented by a link. This means that to represent the starting point of the linked area, we need in fact two values: value x at entry a to represent the link to an existing entry b of the compacted array, and if entry b contains a link to another entry c, we need value Δx to tell which of the entries represented by the link at entry b we are actually linking to. In Fig. 1(b), we have $(x, \Delta x) = (5, 1)$ at the entry 7. The value $\Delta x = 1$ tells that the link at entry 7 points to the second entry of the area that is created by uncompacting link $(3, 0)$ at entry $x = 5$. In principle, we would need a third value to denote the length of the area represented by a link, but a link can also be uncompacted without it (see Sect. 4). However, we still need a boolean value for each entry to tell whether it contains a link or a suffix of the text.

(a)



(b)

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
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	c	9
6	gtaaac	4
7	gtagtaaac	1
8	taaac	5
9	tagtaaac	2

→

entry	suffix	pos
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	→	(3)
6		
7	gtagtaaac	1
8	→	(6)
9		

→

compact suffix array:

entry	suffix	pos
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	→	(3,0)
6	gtagtaaac	1
7	→	(5,1)

Fig. 1. (a) A suffix tree and a CDAWG for text "gtagtaaac" (b) A suffix array for text "gtagtaaac" and construction of a compact suffix array. Arrows in columns with title *suffix* tell that the value(s) in parenthesis in the next column denote a link to another area of the array.

Because the links can be chained, we have to extend the definitions of the compacting and uncompacting operations. A *compacting operation* compacts area $b \ldots b + \Delta$ such that some parts of the area can already be compacted using a compacting operation. A *uncompacting operation* retrieves area $a \ldots a + \Delta$ from an area $b \ldots b + \Delta$ that is compacted using a compacting operation.

Definition 2 *The* compact suffix array (CSA) *of text T of length $n = |T|$ is an array $CA(1 \ldots n')$ of length n', $n' \leq n$, such that for each* entry $1 \leq i \leq n'$, *$CA(i)$ is a triple $(B, x, \Delta x)$, where B is a boolean value to determine whether x denotes a suffix of the text ($B = false$), or whether x and Δx denote a link to an area obtained by a compacting operation ($B = true$). If $B = true$, $x + \Delta x$ denotes the entry of CA where the linked area starts* after *uncompacting $CA(x)$ with an uncompacting operation.*

The relationship between the suffix array and the CSA can be characterized by the fact that the suffix array of the text T can be constructed by uncompacting the links of the CSA. The *optimal* CSA for T is such that its size n' is the smallest possible.

We use $CA(i).B$, $CA(i).x$, and $CA(i).\Delta x$ to denote the value of B, x, and Δx at entry i of the array CA.

3 Constructing a CSA from a Suffix Array

The idea of the algorithm to construct a CSA from a suffix array is shown in Fig. 1(b). First, the areas that can be replaced by a link 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 $A(i) + 1$ could be found by a binary search, but it can also be found in constant time by constructing the reverse A_r of array A in advance; the array A_r is easily created 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 x is marked to be replaced by a link to the area starting at i. 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, that are shown in Fig. 2, work clearly in linear time.

In *MarkLinksToSuffixArray*, the reason for the conditions in the line (8) is to prevent *cyclic links*. For example a simple cyclic link would be that area $\mathcal{A}$ is represented by a link to area $\mathcal{B}$ and area $\mathcal{B}' : \mathcal{B}' \cap \mathcal{B} \neq \emptyset$ is represented by a link to area $\mathcal{A}' : \mathcal{A}' \cap \mathcal{A} \neq \emptyset$ (for example, the CSA of text "ababa" could contain such). Using the array *IsPointed* we choose the areas that can be linked and prevent cyclic links. For example, with the simple cyclic link we choose so that area $\mathcal{A}$ can be replaced by a link to area $\mathcal{B}$, but area $\mathcal{B}'$ can not be replaced by a link to area $\mathcal{A}'$. Unfortunately, this prevents all the other links to the area $\mathcal{A}'' : \mathcal{A}'' \cap \mathcal{A} \neq \emptyset$, too, and results in a structure that is not optimally compressed. Thus, the algorithm does not construct the optimal CSA.

```
MarkLinksToSuffixArray(in A,in n,out A',out n'):
for each 1 ≤ i ≤ n do A'(i).x ← −1;
for each 1 ≤ i ≤ n do A_r(A(i).x) ← i;
i ← 1; n' ← 1;
while i ≤ n do begin
    x ← A_r(A(i).x + 1);
    j ← 0;
    while x + j < n and i + j < n and
            x + j ≠ i and i + j ≠ x and not IsPointed(x + j) and
            A(x + j).x = A(i + j).x + 1 do begin
        if A'(i + j).x < 0 then A'(i + j) ← A(i + j);
        if j > 0 then begin
            A'(x + j) ← (true, i, j);
            IsPointed(i + j) ← true end;
        j ← j + 1 end;
    if j > 1 then begin {update first linked suffix}
        IsPointed(i) ← true;
        A'(x) ← (true, i, 0);
        i ← j + 1 end
    else begin {copy suffix}
        if A'(i).x < 0 then A'(i) ← A(i);
        i ← i + 1 end;
    n' ← n' + 1 end;

CompactMarkedArray(in A',in n,in n',out CA):
j ← 0;
for each 1 ≤ i ≤ n do begin
    if A'(i).Δx = 0 then begin
        CA(j) ← A'(i);
        j ← j + 1;
        mapiToj(i) ← j end
    else mapiToj(i) ← j − 1 end;
for each 1 ≤ i ≤ n' such that CA(i).B = true do
    CA(i) ← (true, mapiToj(i), A'(CA(i).x).Δx);
```

Fig. 2. Algorithm to construct a CSA. For simplicity, arrays A and A' use the same representation as array CA.

4 Searching

The proposed compacting method maintains the important property that the CSA still allows a fast search. The search algorithm works in two phases. First, a binary search is performed over the entries that do not contain a link. This search gives us an area of the array that must be checked. Secondly, checking is done by uncompacting every link of the area, and checking the suffixes formed this way against the pattern. To be able to make a binary search over the CSA, we need some modifications to its original definition.

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

On other words, only every other entry of a BSCSA can be a link. The algorithm *MarkLinksToSuffixArray* in Fig. 2 can be easily modified to handle this limitation. From this on, we assume that a CSA is actually a BSCSA. The search algorithm that utilizes this property is shown in Fig. 3.

```
SearchFromCSA(in P,in CA,in T,in n,in n',out matches):
    {Make two binary searches to find left limit LL and right limit RR}
    {Check the area LL..RR}
    if (LL = RR and CA(LL).B = true) or (LL < RR) then
        matches ← matches + CheckArea(CA, LL, RR, P)
    else if LL = RR then matches ← 1;

CheckArea(in CA, in LL, in RR, in P):
    for each LL ≤ i ≤ RR do begin
        if CA(i).B = true then begin
            LimitSet ← LimitSet ∪ {i};
            UCA ← UnCompactLink(CA, i);
            LimitSet ← LimitSet \ {i};
            if i = LL or i = RR then UCA ← CopyEqual(UCA, P);
            matches ← matches + |UCA| end
        else matches ← matches + 1 end
    return matches;
```

Fig. 3. Search algorithm for a CSA

Algorithm *SearchFromCSA* makes two binary searches to find left and right limits of the area to be checked. The only difference compared to the normal binary search, is the special treatment of the entries containing a link; if the middle of the area is a link, we can choose the entry on its left side instead.

Also, if the last middle point of the search loop is a link, we have to make some additional checkings. After finding left and right limits, we call function *CheckArea* to calculate the sum of the matches.

The reason for the use of the *LimitSet* in *CheckArea* is to prevent function *UnCompactLink* from trying to uncompact cyclic links (i.e. prevents from entering into the areas of the array that cannot be part of the link being uncompacted). Function *UnCompactLink* returns an array that represents the linked area of the original suffix array. The resulting array does not necessarily match totally to the pattern, so function *CopyEqual* returns the matching part. Finally, the number of the matches is incremented by the length of the uncompacted array.

The difficult part of the search algorithm is the function *UnCompactLink*. As the links between the areas of the CSA can be chained, it is natural to write a recursive function to uncompact a link. We first write a procedure that only handles the situation where the linked area does not contain links (i.e. the situation at the end of the recursion). The resulting algorithm is shown in Fig. 4.

SimpleUnCompactLink(in CA, in x):
(1) $y \leftarrow CA(x).x + CA(x).\Delta x$;
(2) $prev \leftarrow CA(x-1).x$;
(3) $i \leftarrow 0$;
(4) **while** $T_{CA(y+i).x+1...n} < T_{CA(x+1).x...n}$ **and**
$\quad T_{CA(y+i).x+1...n} > T_{prev...n}$ **do begin**
(5) $\quad prev \leftarrow CA(y+i).x + 1$;
(6) $\quad UCA(i+1).x \leftarrow prev$;
(7) $\quad i \leftarrow i + 1$ **end**
(8) **return** UCA;

Fig. 4. The simple case of the algorithm to uncompact a link.

It is easy to see that the simple case of the *UnCompactLink* works right in its proper environment: The conditions in the while-loop are for detecting the end of the linked area. If suffix $T_{CA(y+i).x+1...n}$ is greater than previously uncompacted suffix and smaller than $T_{CA(x+1).x...n}$, it must belong to the area being uncompacted. Otherwise it would have been in the place of the suffix $T_{CA(x+1).x...n}$ in the original suffix array, and would have ended up to be in $CA(x+1)$, in which it is not.

Now we can write the full recursive version of the same algorithm, which can handle the chained links; the algorithm is shown in Fig. 5.

In *UnCompactLink*, procedure *AddAndCheck(A,B,inc)* adds the area B to the end of area A and adds *inc* to each added suffix. It also checks whether the end of the linked area is reached with the same conditions as in *SimpleUnCompactLink*. The *LimitSet* is to prevent *UnCompactLink* from trying to uncompact cyclic links; the construction algorithm of Sect. 3 does not create cyclic links, but the uncompacting phase could enter into a cycle while searching the end of the linked area.

The only significant difference between the recursive version and the simple case of the *UnCompactLink* is the treatment of the value Δx: in the recursive version the start of the linked area is not known, and all the links starting after entry x must be checked. When the start of the linked area is found, the end of the area is checked almost similarly in both versions. In the recursive version checking is done in parts: entries that contain a suffix are handled as in the simple case, and entries that contain a link are first uncompacted and then the suffixes in the uncompacted area are handled as in the simple case.

```
UnCompactLink(in CA, in x):
y ← CA(x).x; Δx ← CA(x).Δx; i ← 0;
while (y + i) ∉ LimitSet do begin
    if CA(y + i).B = true then begin {y + i contains a link}
        LimitSet ← LimitSet ∪ {y + i};
        UCA ← UnCompactLink(CA, y + i);
        LimitSet ← LimitSet \ {y + i};
        if AddAndCheck(Area, UCA(Δx + 1 ... |UCA|),1) then break;
        Δx ← max(0, Δx − |UCA|)) end
    else begin {y + i contains a suffix}
        if Δx > 0 then Δx ← Δx − 1
        else if AddAndCheck(Area, CA(y + i).x, 1) then break end;
    i ← i + 1 end;
return Area;
```

Fig. 5. Recursive algorithm to uncompact a link.

4.1 The Search Time

The first part of the algorithm *SearchFromCSA* of Fig. 3, that finds the area to be checked, works in time $O(|P| \log n')$, as the pattern P is compared to each suffix of the binary search. We can, however, store LCP values for the entries that contain suffixes (see Sect. 5.1). With this information, the first part of the algorithm *SearchFromCSA* can be implemented to work in time $O(|P| + \log n')$.

The time needed for the recursive subprocedure *UnCompactLink* is presented in the following lemma:

Lemma 4 *The average time needed to uncompact a link of CSA is* $O((\frac{2n-n'}{n'})^2 \log n)$, *where* $n = |T|$ *and* n' *is the length of the CSA.*

Proof. The structure of the CSA guarantees that at most every other entry contains a link. We start the average case analysis by considering the worst case, i.e. every other entry contains a link. This gives us an equation $\frac{n'}{2}x + \frac{n'}{2} = n$ to the average length of the linked area x. The value $\frac{n'}{2}x = n - \frac{n'}{2}$ is the sum of the lengths of the linked areas. If we assume that every entry of the original suffix array is equally likely to be the start of a linked region, we can write the probability of a link path of length k as $\mathbb{P}($"*link path of length* k"$)= (\frac{n-\frac{n'}{2}}{n})^{k-1}(1 - \frac{n-\frac{n'}{2}}{n}) = (1 - \frac{n'}{2n})^{k-1}\frac{n'}{2n}$ (i.e. $k-1$ times a link leads to an entry containing a link and the last link leads to an entry containing a suffix). This is a geometric probability distribution and its expectation value is $\mathbb{E}($"*link path of length* k"$) = \frac{2n-n'}{n'}$.

As each link represents on the average $\frac{2n-n'}{n'}$ suffixes and the length of the link path to each such suffix is on the average $\frac{2n-n'}{n'}$, and every such suffix is compared to some other suffixes (time $O(\log n)$), the total time used to uncompact the suffixes presented by a link is on the average $O((\frac{2n-n'}{n'})^2 \log n)$.

In addition to uncompacting the suffixes represented by a link, the algorithm of Fig. 5 spends time finding the start of the linked area. This must be done only when an area is linked inside an already linked area. In such cases, there is on average $\frac{2n-n'}{2n'}$ suffixes to uncompact to find the start of the linked area, because in the average case a link points in the middle of the already linked area. As the length of the link path is on the average $\frac{2n-n'}{n'}$, it takes $O(\frac{1}{2}(\frac{2n-n'}{n'})^2 \log n)$ to find the start of the linked area.

The total time to uncompact a link is therefore $O(\frac{3}{2}(\frac{2n-n'}{n'})^2 \log n)$. □

Theorem 5 *The average time needed to search all occurrences of pattern P from CSA of text T is $O((\frac{2n-n'}{n'})^2(|P| + k\log n))$, where $n = |T|$, n' is the length of the CSA, and k is the number of occurrences.*

Proof. The binary search of the area of possible occurrences takes time $O(|P| + \log n')$. If k is the number of occurrences, there will be at most $k/3$ links to uncompact (when each link represents the minimum two suffixes). Therefore the result of Lemma 4 must be multiplied by $k/3$. In addition, the suffixes represented by the first and the last links must be checked against the pattern P (other suffixes already match due to the binary search). The total search time is the sum of these three factors. □

Theorem 6 *The average time needed to determine whether there is an occurrence of pattern P inside text T, using CSA is $O((\frac{2n-n'}{n'})^2(|P| + \log n))$, where $n = |T|$ and n' is the length of the CSA.*

Proof. There is at most one link to uncompact, because if there were two (or more) links to uncompact, there would be a suffix between them that matches P. Therefore we get the time $O((\frac{2n-n'}{n'})^2(|P| + \log n))$. □

If $n' = \Theta(n)$, the value $(\frac{2n-n'}{n'})^2$ can be considered as a constant, and the results of Lemma 5 and Theorem 6 can be improved. This is the case for typical texts, as can be seen from the experimental results in Sect. 6.

5 Practical implementation

5.1 Space Requirement

Although a CSA has fewer entries than the corresponding suffix array, an entry of a CSA takes more space than an entry of a suffix array. An entry of a suffix array contains only one integer value, while in a CSA there are two integer values and one boolean value in an entry (triple $(B, x, \Delta x)$). This can be overcome by the fact that we can limit the length of the linked region at the cost of slightly smaller compression. If we limit the length of the linked region so that $0 \leq \Delta x \leq 255$, we can use one byte to represent the value of Δx[1]. Also the boolean value B can be embedded into the value x at the cost of only being able to handle texts that are smaller than 2^{31} bytes.

Fact 7 *An entry of a CSA takes* $5/4$ *times the space of an entry of a suffix array.*

Since the value Δx is only needed for the entries that contain a link, the wasted space can be used to store the LCP information of consecutive suffixes of the binary search. As each suffix has two possible preceding suffixes in a binary search, there are two values to be saved for each suffix. There was suggested in [9], that saving only the maximum of these two values is adequate, because the other value can be achieved during the search. However, we need one bit to tell which value is actually saved. There remains 7 bits to save the LCP information, i.e the LCP values must be in the range $0 \ldots 127$. In an average case this is an acceptable limitation[2]. The special case when a LCP value does not fit into a byte can be handled separately (costing in search times).

Fact 8 *An entry of a suffix array with the LCP information of consecutive suffixes of the binary search takes the same space as an entry of a CSA with the LCP information.*

5.2 An Implementation with Improved Worst Case Bounds

Although the algorithm in Fig. 5 has reasonably good average case bounds, its worst case behavior is a problem; finding the start of the linked area may require uncompacting the whole array.

Therefore we make the following modifications to the structure of the CSA and construction algorithm of Fig. 2:

---

[1] We assume an environment, where an integer takes 32 bits and consists of four bytes.

[2] The largest value of a LCP of two suffixes of a text T equals to the length of the *longest repeat* in T, i.e. the longest substring of T that occurs at least twice. As the probability of a repeat of length m is $\frac{1}{|\Sigma|^m}$ and there are approximately $|T|^2$ places to start the repeat, we can write the expectation value $\mathbb{E}($"*number of repeats of length* m"$) \approx \frac{|T|^2}{|\Sigma|^m}$. If the longest repeat is unique, its length m_l should satisfy $1 \approx \frac{|T|^2}{|\Sigma|^{m_l}}$, and therefore $m_l \approx 2 \log_{|\Sigma|} |T|$ (see the more detailed analysis in [7, 12]). Thus, the values of LCPs are expected to be in the range $0 \ldots 127$.

- The length of the linked area is added to the structure of the CSA. That is, each $CA(i)$ in Definition 2 contains $(B, x, \Delta x, \Delta)$, where Δ is the length of the linked area (i.e. $\Delta = \Delta_2 + 1$, where Δ_2 denotes the value Δ introduced in Sect. 2).
- The length of the linked area is limited by a constant $\mathcal{C}$ by adding condition $j < \mathcal{C}$ in while-loop at line (7) of procedure *MarkLinksToSuffixArray* in Fig. 2. Also, the value of j must be copied to the value $A'(x).\Delta$ at line (17).
- The length of the link path is limited by a constant $\mathcal{D}$ by using array $pathdepth(1 \ldots n)$ in procedure *MarkLinksToSuffixArray*. The array entries are initialized to zero and condition $pathdepth(i + j) < \mathcal{D}$ is added in the line (7). The array $pathdepth$ is updated by adding $pathdepth(x + j) \leftarrow pathdepth(i + j) + 1$ in line (12) and $pathdepth(x) \leftarrow pathdepth(i) + 1$ in line (16).

Storing the value Δ similarly as Δx (using 8 bits) would imply that an entry of the CSA takes 6/5 times the size of an entry of the suffix array with LCP information. This is impractical, and therefore we store both values using only 4 bits (i.e. $\mathcal{C} = 15$, $\Delta \leq \mathcal{C}$, and $\Delta x \leq \mathcal{C}$). This does not affect the compression significantly, as can be seen from the experimental results in Sect. 6.

With the value Δ in the CSA, we can now replace the algorithm *UnCompactLink* by a more efficient version, that is shown in Fig. 6.

```
UnCompactLinkPractical(in CA, in x):
y ← CA(x).x; Δx ← CA(x).Δx; Δ ← CA(x).Δ; i ← 0;
while Δx > 0 do begin
    if CA(y + i).B = false then begin
        i ← i + 1; Δx ← Δx − 1 end
    else begin
        if CA(y + i).Δ ≤ Δx then begin
            Δx = Δx − CA(y + i).Δ; i ← i + 1 end
        else break end end;
while true do begin
    if CA(y + i).B = true then begin {y + i contains a link}
        UCA ← UnCompactLink(CA, y + i);
        Add(Area, UCA(Δx + 1 ... min(|UCA|, Δx + Δ)),1);
        Δx ← 0; end
    else {y + i contains a suffix}
        Add(Area, CA(y + i).x, 1);
    if |Area| ≥ Δ then break;
    i ← i + 1 end;
return Area;
```

Fig. 6. A version of *UnCompactLink* with improved worst case bounds.

Theorem 9 *The algorithm in Fig. 6 uncompacts a link of the CSA using* $O(\mathcal{C}^2(\mathcal{D} - \log_2 \mathcal{C}))$ *time in the worst case, where* $\mathcal{C}$ *is a constant limiting the length of the linked area and* $\mathcal{D}$ *is a constant limiting the length of the link path.*

Proof. The recursive search tree of the worst case is visualized in Fig. 7. The first

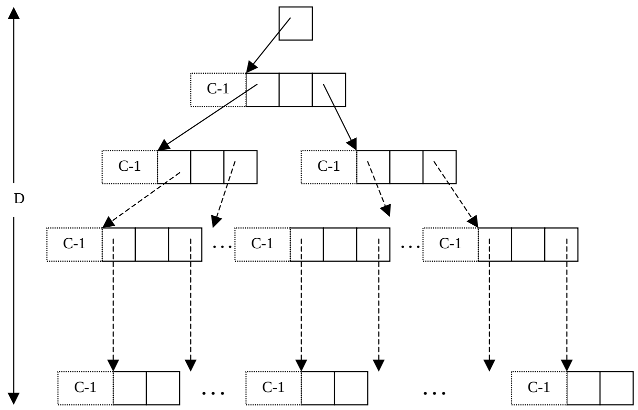


Fig. 7. The recursive search tree to uncompact a link in the worst case.

call to *UnCompactLink* requires to skip at most $\mathcal{C}-1$ suffixes (when $\Delta x = \mathcal{C}-1$). After that, there is a link, a suffix, and a link to represent the linked area, so that the tree will grow as much as possible and code as little information as possible. Because a link can represent at most $\mathcal{C}$ suffixes, there is a limit to the growth of the tree. On the other hand, there could be link paths that do not code any information. As we have limited the length of the link path by a constant $\mathcal{D}$, the worst case is achieved when the links at the last level of the search tree each represent the minimum two suffixes that are located at the end of the link path of length $\mathcal{D}$.

The amount of widening levels (denoted by x) in the worst case search tree can be solved from the following equation:

$$\sum_{i=0}^{x} 2^i + 2 \cdot 2^{x+1} = \mathcal{C}, \tag{1}$$

where the summation denotes the suffixes at levels $1 \ldots x$ and the last term denotes the suffixes at the end of the link path. The equation (1) has a solution $x = \log_2 \frac{\mathcal{C}+1}{3} - 1$. We have to limit to the values of $\mathcal{C}$ that result a whole number solution to the value of x (for other values, the worst case search tree of Fig. 7 would be unbalanced, and thus slightly harder to analyze).

To calculate the time needed to uncompact a link, we have to multiply each node of the search tree with $\mathcal{C}-1$. Therefore the time requirement is:

$$(\mathcal{C}-1)(\sum_{i=0}^{x} 2^i + 2^{x+1}(\mathcal{D}-x)), \tag{2}$$

which can be written as follows by substituting the above value for x:

$$(\mathcal{C}-1)(\frac{\mathcal{C}+1}{3} - 1 + \frac{\mathcal{C}+1}{3}(\mathcal{D} - \log_2 \frac{\mathcal{C}+1}{3} + 1))). \tag{3}$$

□

As a corollary of Theorem 9, we have $O(|P| + \log n')$ worst case search time from the CSA (when LCP values are stored in integer fields). Although, the constant factor is quite big compared to the original suffix array. The experimental results in next section show that the constants $\mathcal{C}$ and $\mathcal{D}$ can be set relatively small without significant loss in compression.

6 Experimental results

Two important properties of the CSA were tested: how much CSA saves space as compared to the suffix array, and how much the search times suffer from compression.

The space requirements of some files from the Calgary and Canterbury corpora [1] are shown in Table 1. The SA_LCP means the size of a suffix array with the LCP information, i.e. five times the size of the text. The CSA is constructed as described in Sect. 5 using values $\mathcal{C} = 15$ and $\mathcal{D} = 4$ (the values in parentheses in the last column of Table 1 show the reduction percentage when $\mathcal{C} = 255$ and $\mathcal{D}$ is not used). As can be seen, the corresponding CSA takes about half the size of the suffix array with two texts: *world192.txt* and *progp* (Pascal-program code). This is quite good result as normal compression routine like *zip* compresses these texts by 71% and 77%. The text *book1* does not have much repetitiveness and is therefore compressed poorly even with *zip* (59%). The worst text to the CSA is *e.coli* containing DNA, though it compresses well with *zip* (71%). This is due to the nature of the repetitiveness; e.g. string "aggggt" may occur frequently in DNA, but so will occur strings "cggggt", "gggggt", and "tggggt". When a suffix array is constructed, suffixes starting with "aggggt" will be before suffixes starting with "cggggt", "gggggt", and "tggggt". Now, suffixes starting with "ggggt" cannot be represented as links to any of these suffixes starting with "aggggt", "cggggt", "gggggt", or "tggggt", because they will occur in different order. This is in contrast to the word "procedure" in *progp*; there will be no other words containing word "rocedure" and so all suffixes starting with "rocedure" can be replaced by a link to the area of suffixes starting with "procedure".

The search times are compared in Table 2. The total time to search 10000 patterns is reported for each text. For natural language texts, the patterns are

Table 1. Space requirement

text	|text|	SA_LCP	CSA	reduction %
book1	768 771	3 843 855	2 502 785	35% (36%)
e.coli	4 638 690	23 193 450	18 722 175	19% (19%)
progp	49 379	246 895	129 560	48% (52%)
world192.txt	2 473 400	12 367 000	6 590 370	47% (52%)

Table 2. Search times

text	|text|	occurrences	SA_LCP (ms)	CSA (ms)
book1	768 771	4 466	27	39
book1	768 771	530 950	32	112
e.coli	4 638 690	10 000	34	39
e.coli	4 638 690	6 328 248	90	1054
progp	49 379	1 224	21	49
progp	49 379	34 151	21	81
world192.txt	2 473 400	5 632	34	47
world192.txt	2 473 400	1 283 892	44	278

chosen randomly from different sources. For *e.coli*, all patterns of length 6 are generated and they are combined with some patterns of length 7 to get the total amount of 10000. Each search is repeated 10, 100, or 1000 times depending on the size of the text. The first line on each text denotes the time needed to determine whether there is an occurrence of the pattern (the column *occurrences* reports the amount of patterns that were found). The second line on each text denotes the time needed to report all the occurrences. Times are milliseconds on a Pentium III 700 MHz machine with 774 MB main memory running Linux operating system.

The search algorithms use LCP values, and thus work in $O(|P|+\log|T|)$ time in the average case. The LCP values were calculated using a simple algorithm that uses $O(|T|\log|T|)$ time in the average case (the worst case is not a problem, because we have limited the lengths of the LCPs to be smaller than 128).

It can be seen that with the basic string matching query, the CSA is not significantly slower than the suffix array. As predicted, the more occurrences there are, the more time takes searching from the CSA. The reason why the search time also increases in the suffix array, is that matches were reported (not printed, but calculated one by one) for the sake of comparison.

7 Conclusion

We have considered an efficient way to compact a suffix array to a smaller structure. A new search algorithm has been developed for the string-matching problem using this compact suffix array. It has been shown that a basic string-matching query can be answered in average time $O(|P| + \log |T|)$ on typical texts. The same bound can be achieved in the worst case, although with a larger constant factor and smaller compression.

The algorithm in Sect. 3 that constructs a CSA from a suffix array does not necessarily give the smallest CSA, because the method used for preventing cycles, can also prevent some non-cyclic links. It is an open problem to develop an efficient algorithm to construct the smallest possible CSA (a Union-Find approach for preventing cycles could be possible). Another open problem is to construct a CSA straight from the text. This would save space at construction time, as our algorithm needs space for two suffix arrays and some additional structures during the construction.

Acknowledgement

I wish to thank Esko Ukkonen and Juha Kärkkäinen for their comments and advises.

References

1. R. Arnold and T. Bell, *A corpus for the evaluation of lossless compression algorithms*, Proceedings of the Data Compression Conference, 1997, pp. 201-210. http://corpus.canterbury.ac.nz
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, *The smallest automaton recognizing the subwords of a text*, Theor. Comp. Sci., 40 (1985), pp. 31-55.
3. A. Blumer, J. Blumer, D. Haussler, and R. McConnell, *Complete inverted files for efficient text retrieval and analysis*, Journal of the ACM, 34:3 (1987), pp. 578-595.
4. A. Blumer, D. Haussler, A. Ehrenfeucht, *Average sizes of suffix trees and dawgs*, Discrete Applied Mathematics, 24 (1989), pp. 37-45.
5. M. Crochemore, *Transducers and repetitions*, Theor. Comp. Sci., 45 (1986), pp. 63-86.
6. M. Crochemore and Renaud Vérin, *Direct Construction of Compact Directed Acyclic Word Graphs*, In Proc. of the CPM, LNCS 1264 (1997), pp. 116-129.
7. Erdös and Rényi, *On a new law of large numbers*, J. Anal. Math. 22 (1970), pp. 103-111.
8. 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.
9. U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput., 22 (1993), pp. 935-948.
10. E. M. McCreight, *A space economical suffix tree construction algorithm*, Journal of the ACM, 23 (1976), pp. 262-272.
11. E. Ukkonen, *On-line construction of suffix-trees*, Algorithmica, 14 (1995), pp. 249-260.
12. M. S. Waterman, *Introduction to Computational Biology*, Chapman & Hall, University Press, Cambridge, UK, 1995, pp. 263-265.
13. P. Weiner, *Linear pattern matching algorithms*, In Proc. IEEE 14th Annual Symbosium on Switching and Automata Theory, 1973, pp. 1-11.