Two Space Saving Tricks for Linear Time LCP Array Computation*

Giovanni Manzini

Dipartimento di Informatica, Università del Piemonte Orientale,
I-15100 Alessandria, Italy.
manzini@mfn.unipmn.it.

Abstract. In this paper we consider the linear time algorithm of Kasai *et al.* [6] for the computation of the Longest Common Prefix (LCP) array given the text and the suffix array. We show that this algorithm can be implemented without any auxiliary array in addition to the ones required for the input (the text and the suffix array) and the output (the LCP array). Thus, for a text of length n, we reduce the space occupancy of this algorithm from $13n$ bytes to $9n$ bytes.
We also consider the problem of computing the LCP array by "overwriting" the suffix array. For this problem we propose an algorithm whose space occupancy can be bounded in terms of the empirical entropy of the input text. Experiments show that for linguistic texts our algorithm uses roughly $7n$ bytes. Our algorithm makes use of the Burrows-Wheeler Transform even if it does not represent any data in compressed form. To our knowledge this is the first application of the Burrows-Wheeler Transform outside the domain of data compression.
The source code for the algorithms described in this paper has been included in the *lightweight suffix sorting* package [13] which is freely available under the GNU GPL.

1 Introduction

The *suffix array* [11] is a simple and elegant data structure used for several fundamental string matching problems involving both linguistic texts and biological data. The vitality of this data structure is proven by the large number of suffix array construction algorithms developed in the last two years [1,5,7,8,14]. The suffix array of a text $\mathtt{t}[1, n]$ is the lexicographically sorted list of all its suffixes. The suffix array is often used together with the *Longest Common Prefix array*, LCP array from now on, which contains the length of the longest common prefix between every pair of lexicographically consecutive suffixes. The LCP information can be used to speed up suffix array algorithms and to simulate the more powerful, but more resource consuming, suffix tree data structure [6,11].

In [6] Kasai et al. describe a simple (13 lines of C code) and elegant linear time algorithm for computing the LCP array given the text and the suffix array.

* Partially supported by the Italian MIUR projects "Algorithmics for Internet and the Web" and "Technologies and Services for Enhanced Content Delivery".

T. Hagerup and J. Katajainen (Eds.): SWAT 2004, LNCS 3111, pp. 372–383, 2004.
© Springer-Verlag Berlin Heidelberg 2004

This was an important result for several reasons. First, although many suffix array construction algorithms can be modified to return the LCP array as well, this is not true for every algorithm. Having decoupled the two problems allows one to choose the suffix array construction algorithm which better suits his/her needs without the constraint of considering only those algorithms which also provide the LCP array. Moreover, in some applications one may need the LCP array later than the suffix array: if one has to compute them simultaneously some temporary storage must be used for the LCP array.

The only drawback of the algorithm of Kasai et al. is its large space occupancy. Assuming a "real world" model in which each text symbol takes one byte and each suffix array or LCP array entry takes 4 bytes, the algorithm of Kasai et al. uses $13n$ bytes, where n is the length of the input text. Considering that the output of the computation (text, suffix array, and LCP array) takes $9n$ bytes, we have a $4n$ bytes overhead which is a serious issue since it is nowadays common to work with files hundreds of megabytes long.

In this paper we present a modified version of the algorithm of Kasai et al. which only uses $9n$ bytes of storage. Our algorithm, called Lcp9, runs in linear time and has the same simplicity and elegance of the original algorithm. Experiments with several files of different size and structure show that Lcp9 is only 5%–10% slower than the algorithm of Kasai et al.[1]

In our "real world" model, a space occupancy of $9n$ bytes is optimal if we assume that at the end of the computation we need the text, the suffix array, and the LCP array. However, this is no longer true if one is interested *only* in the LCP array, that is, if at the end of the computation we no longer need the suffix array. In this case, the space initially used for storing the suffix array can be reused during the computation and for the storage of the LCP array. In this scenario we can aim to a space occupancy as low as $5n$ bytes. The problem of computing the LCP array discarding the suffix array has applications in the fields of string matching, data compression and text analysis. For example, using the algorithm described in [6, Sect. 5] with a single pass over the LCP array we can simulate a post order visit of the suffix tree of the text $\mathtt{t}$. In some applications, for example for the construction of the compression booster described in [3], such visit does not need the information stored in the suffix array.

If we only need the LCP array, even the $9n$ bytes space occupancy of algorithm Lcp9 becomes the space bottleneck of the whole computation since for the construction of the suffix array there are "lightweight" algorithms [1,14] which only use $(5+\epsilon)n$ bytes with $\epsilon \ll 1$. In this paper we address this issue proposing a simple linear time algorithm, called Lcp6, which computes the LCP array by "overwriting" the suffix array. The space used by Lcp6 depends on the regularity of the input text $\mathtt{t}[1,n]$ and can be bounded in terms of the k-th order empirical entropy of $\mathtt{t}$. If $\mathtt{t}$ is highly compressible the space occupancy of Lcp6 can be as small as $6n$ bytes. Vice versa, if $\mathtt{t}$ is a "random" string the space

[1] Recently, we found out that in [9] Mäkinen describes a space-economical version of Kasai et al. algorithm which only uses $9.125n$ bytes. We will report on Mäkinen's algorithm in the full paper.

required by our algorithm can be as large as $10n$ bytes. Note however that in the first step of Lcp6 we can evaluate exactly how much space it will need: if such space turns out to be larger than $9n$ bytes we can quit Lcp6 and compute the LCP array using Lcp9. Thus, combining Lcp6 and Lcp9 we get an algorithm with a space occupancy between $6n$ and $9n$ bytes. The experimental results show that for linguistic texts, source code, and xml/html documents Lcp6 always uses less than $8n$ bytes and for the largest files it often uses less than $7n$ bytes. For DNA sequences Lcp6 uses between $8n$ and $9n$ bytes, and—not surprisingly—for compressed files it uses close to $10n$ bytes.

We point out that our algorithms only have a "practical" interest: from the theoretical point of view their working space of $\Theta(n \log n)$ bits is not optimal. Indeed, the optimal space/time tradeoff can be obtained combining the results in [4] and [15] which allow one to build the suffix array and LCP array in linear time using $O(n)$ bits of auxiliary storage. Unfortunately the algorithms in [4,15] are quite complex and it is still unclear whether they will lead to competitive practical algorithms.

2 Background and Notation

Let Σ denote a finite ordered alphabet. Without loss of generality, in the following we assume that Σ consists of the integers $1, 2, \ldots, |\Sigma|$. Let $\mathtt{t}[1,n]$ denote a text over Σ. For $i = 1, \ldots, n$ we write $\mathtt{t}[i,n]$ to denote the suffix of $\mathtt{t}$ of length $n-i+1$ that is $\mathtt{t}[i,n] = \mathtt{t}[i]\mathtt{t}[i+1]\cdots\mathtt{t}[n]$.

The suffix array [11] for $\mathtt{t}$ is the array $\text{SA}[1,n]$ such that $\mathtt{t}[\text{SA}[1],n]$, $\mathtt{t}[\text{SA}[2],n]$, ..., $\mathtt{t}[\text{SA}[n],n]$ is the list of suffixes of $\mathtt{t}$ sorted in lexicographic order. To define unambiguously the lexicographic order of the suffixes it is customary to logically append at the end of $\mathtt{t}$ a special end-of-string symbol # which is smaller than any symbol in Σ. For example, for $\mathtt{t} = \mathtt{baaba}$, $\text{SA} = [5, 2, 3, 4, 1]$ since $\mathtt{t}[5,5] = \mathtt{a}$ is the suffix with the lowest lexicographic rank, followed by $\mathtt{t}[2,5] = \mathtt{aaba}$, followed by $\mathtt{t}[3,5] = \mathtt{aba}$ and so on.

The rank array $\text{RANK}[1,n]$ of $\mathtt{t}$ is the inverse of the suffix array. That is, $\text{RANK}[i] = j$ if and only if $\text{SA}[j] = i$. Note that $\text{RANK}[i]$ is the rank of the suffix $\mathtt{t}[i,n]$ in the lexicographic order of the suffixes. The LCP array $\text{LCP}[1,n]$ of $\mathtt{t}$ is an array such that $\text{LCP}[i]$ contains the length of the longest common prefix between the suffix $\mathtt{t}[\text{SA}[i],n]$ and its predecessor in the lexicographic order (which is $\mathtt{t}[\text{SA}[i-1],n]$). Note that $\text{LCP}[1]$ is undefined since $\mathtt{t}[\text{SA}[1],n]$ is the lexicographically smallest suffix and therefore it has no predecessor.

Finally, we define the RANKNEXT map such that:

$$\text{RANKNEXT}(i) = \text{RANK}[\text{SA}[i]+1], \qquad \text{for } i = 1, \ldots, n,\ i \neq \text{RANK}[n]. \quad (1)$$

$\text{RANKNEXT}(i)$ is the rank of the suffix $\mathtt{t}[\text{SA}[i]+1,n]$, that is, the rank of the suffix obtained removing the first character from the suffix of rank i. Note that $\text{RANKNEXT}(\cdot)$ is not defined for $i = \text{RANK}[n]$ because in this case $\mathtt{t}[\text{SA}[i]+1,n]$ is the empty string.

```
                       F           L
mississippi#           # mississipp i
ississippi#m           i #mississip p
ssissippi#mi           i ppi#missis s
sissippi#mis           i ssippi#mis s
issippi#miss           i ssissippi# m
ssippi#missi    ==>    m ississippi #
sippi#missis           p i#mississi p
ippi#mississ           p pi#mississ i
ppi#mississi           s ippi#missi s
pi#mississip           s issippi#mi s
i#mississipp           s sippi#miss i
#mississippi           s sissippi#m i
```

Fig. 1. The Burrows-Wheeler Transform for t = `mississippi`. The output of the transform is the last column of the sorted matrix $\mathcal{M}$, i.e., bwt = `ipssm#pissii`.

2.1 The Burrows-Wheeler Transform

In 1994, Burrows and Wheeler [2] introduced a transform that turns out to be very elegant in itself and extremely useful for data compression. Given a string t, the transform consists of three basic steps (see Fig. 1): (1) append to the end of t a special symbol # smaller than any other symbol in Σ; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string t#, sorted in lexicographic order; (3) construct the transformed text bwt by taking the last column of $\mathcal{M}$. Notice that every column of $\mathcal{M}$, hence also the transformed text bwt, is a permutation of t#.

If the input string t has length n, the transformed string bwt has length $n+1$ because of the presence of the # symbol. In the following we assume that the transformed string is stored in an array indexed from 0 to n. For example, in Fig. 1 we have bwt[0] = `i`, bwt[5] = #, bwt[11] = `i`. Using this notation and observing that the rows of the matrix $\mathcal{M}$—up to symbol # on each row—are precisely the suffixes of t in lexicographic order , the computation of bwt given t and the suffix array can be easily accomplished with the code of Fig. 2 (procedure Sa2Bwt). From bwt we can always recover t. The inverse transform is based on the following remarkable property. Let $F[0,n]$ and $L[0,n]$ denote respectively the first and last column of the matrix $\mathcal{M}$ (hence, $L \equiv$ bwt). Then, for any $\sigma \in \Sigma$ we have that the k-th occurrence of σ in F corresponds to the k-th occurrence of σ in L. For example, in Fig. 1 we have that the second `i` in F (that is, $F[2]$) corresponds to the second `i` in L (that is, $L[7]$) since they both are the eighth symbol of `mississippi`. Similarly, the third `s` in F ($F[10]$) corresponds to the third `s` in L ($L[8]$) since they both are the sixth symbol of `mississippi`.

Assume now that the character $F[j]$ corresponds to $L[i]$. This means that row i of $\mathcal{M}$ consists of a (rightward) cyclic shift of row j. Because of the relationship between rows of $\mathcal{M}$ and suffixes of t this is equivalent to stating that the i-th suffix in the lexicographic order is equal to the j-th suffix with the first symbol removed. In terms of the map RankNext defined by (1) we have RankNext$(j) = i$. From this latter relationship it follows that from bwt we can obtain the RankNext map. Indeed, we only need to scan the array bwt (which coincides with column L) finding, for $i = 1, \ldots, n$ the character $F[j]$ corresponding to bwt$[i] \equiv L[i]$. The resulting code is given in Fig. 2 (procedure Bwt2RankNext). Note that column F is not represented explicitly (since it would take $O(n)$ space). Instead we use the array `count`$[1, |\Sigma|]$: at the beginning of the i-th iteration `count`$[k]$ contains the number of occurrences in t of the characters $1, 2, \ldots, k-1$ plus the number of occurrences of character k in bwt$[0] \cdots$ bwt$[i-1]$.

Given the RankNext map and the array bwt, we can recover t as follows. The position of the end-of-string symbol # in bwt gives us Rank(1), that is, the position of t$[1, n]$ in the suffix array. By (1), setting $i =$ Rank(j) we get

$$\textsc{RankNext}(\textsc{Rank}(j)) = \textsc{Rank}(\textsc{Sa}[\textsc{Rank}(j)] + 1) = \textsc{Rank}(j + 1). \qquad (2)$$

Hence, given RankNext and Rank(1) we can generate the sequence of values Rank(2), Rank(3), ..., Rank(n) using the recurrence (2). From the sequence Rank(1), Rank(2), ..., Rank(n) we recover t using the relationship t$[i] =$ bwt[Rank$(i + 1)$]. The corresponding code is shown in Fig. 2 (procedure RankNext2Text).

We conclude this section observing that from the sequence Rank(1), ..., Rank(n) we can also recover the suffix array since $k =$ Rank(i) implies Sa$[k] = i$. The corresponding code is shown in Fig. 2 (procedure RankNext2SuffixArray). Note that in RankNext2SuffixArray as soon as we have read `rank_next[k]` in Line 3 that entry is no longer needed. Therefore, if we replace Line 4 with the instruction `rank_next[k] = i++;` we get a procedure which stores the suffix array entries in the array `rank_next` overwriting the old content of the array (the RankNext map). This property will be used in Section 4.

2.2 The Algorithm of Kasai et al.

The algorithm of Kasai et al. (algorithm Lcp13 from now on) takes as input the text t$[1, n]$ and the corresponding suffix array Sa$[1, n]$ and returns the LCP array. For $i = 1, \ldots, n$ let ℓ_i denote the LCP between t$[i, n]$ and the suffix immediately preceding it in the lexicographic order (ℓ_i is undefined when t$[i, n]$ is the lexicographically smallest suffix). The algorithm Lcp13 computes the LCP values in the order $\ell_1, \ell_2, \ldots, \ell_n$.

The code of Lcp13 is shown in Fig. 3. As a first step (Line 1) the algorithm computes the rank array Rank$[1, n]$. Then, at the i-th iteration of the main loop (Lines 3–13) Lcp13 computes ℓ_i as follows. At Line 4 the value Rank$[i]$ is stored in the variable `k`. If Rank$[i] = 1$ then t$[i, n]$ is the smallest suffix in the lexicographic order and ℓ_i is undefined (we set it to -1 at Line 5). If Rank$[i] > 1$, we compute $j =$ Sa[Rank$[i] - 1$] (Line 7). t$[j, n]$ is the suffix preceding t$[i, n]$ in the lexicographic order, hence ℓ_i is the longest common prefix between t$[i, n]$ and t$[j, n]$.

The crucial observation, which ensures that Lcp13 runs in $O(n)$ time, is that whenever ℓ_i and ℓ_{i-1} are both defined we have $\ell_i \geq \ell_{i-1} - 1$ (Theorem 1 in [6]).

Procedure Sa2Bwt

```
bwt[0]=t[n];
for(i=1;i<=n;i++) {
  if(sa[i] == 1)
     bwt[i]='#';
  else
     bwt[i]=t[sa[i]-1];
}
```

Procedure Bwt2RankNext

```
for(i=0;i<=n;i++) {
  c = bwt[i];
  if(c == '#')
    eos_pos = i;
  else {
     j = ++count[c];
     rank_next[j]=i;
  }
}
return eos_pos;
```

Procedure RankNext2Text

```
k = eos_pos; i=1;
do {
  k = rank_next[k];
  t[i++] = bwt[k];
} while(k!=0);
```

Procedure RankNext2SuffixArray

```
k = eos_pos; i=1;
while(k!=0) {
  nextk = rank_next[k];
  sa[k] = i++;
  k = nextk;
}
```

Fig. 2. Algorithms related to the Burrows-Wheeler Transform. Procedure Sa2Bwt computes the array `bwt` given the text `t` and the suffix array `sa`. Procedure Bwt2RankNext stores in `rank_next` the RankNext map and returns the value Rank(1). The procedure uses the auxiliary array `count`$[1, |\Sigma|]$ which initially contains in `count[i]` the number of occurrences in `bwt` (and therefore in `t`) of the characters $1, \ldots, i-1$. Procedure RankNext2Text recovers the text `t` given the arrays `bwt` and `rank_next` and the value Rank(1) stored in `eos_pos`. Procedure RankNext2SuffixArray computes the suffix array given `rank_next` and the value Rank(1) stored in `eos_pos`.

Procedure Lcp13

```
for(i=1;i<=n;i++) rank[sa[i]] = i;
h=0;
for(i=1;i<=n;i++) {
  k = rank[i];
  if(k==1) lcp[k]=-1;
  else {
    j = sa[k-1];
    while(i+h<=n && j+h<=n && t[i+h]==t[j+h]):
      h++;:
    lcp[k] = h;
  }
  if(h>0) h--;
}
```

Fig. 3. Algorithm of Kasai et al. for the linear time computation of the LCP array. The algorithm takes as input the text `t` and the suffix array `sa` and stores in `lcp` the LCP array. The algorithm uses an auxiliary array `rank` to store the rank array (which is the inverse of the suffix array).

To use this property Lcp13 maintains the invariant that at the beginning of the i-th iteration the variable h contains the value $\ell_{i-1} - 1$. Hence, ℓ_i is computed comparing $\mathtt{t}[i, n]$ and $\mathtt{t}[j, n]$ starting from position h (Lines 8–9). Note that at Line 10 Lcp13 stores ℓ_i in $\text{LCP}[\text{RANK}[i]]$ since the definition of LCP array states that $\text{LCP}[t]$ contains the LCP between $\mathtt{t}[\text{SA}[t], n]$ and $\mathtt{t}[\text{SA}[t-1], n]$.

In our "real world" model, algorithm Lcp13 requires n bytes for the array t and $4n$ bytes for each one of the arrays SA, RANK, and LCP. Therefore its peak space occupancy is $13n$ bytes.

3 LCP Computation in 9n Bytes of Storage

In this section we show how to modify the algorithm of Kasai et al. for computing the LCP array in linear time without using any auxiliary array. As a result we get an algorithm which only uses $9n$ bytes of storage. Our approach consists in using the array `lcp` for storing both "rank information" and "LCP information". Initially the array contains only "rank information". Then, at each iteration of the main loop one item of rank information is used and replaced by one item of LCP information. At the end of the computation the array `lcp` only contains LCP information.

Our starting point is the observation that algorithm Lcp13 (Fig. 3) uses the rank information only in Line 4 where, during the i-th iteration of the main loop, the algorithm retrieves the value $\text{RANK}(i)$. Therefore, Lcp13 uses the sequence of rank values $\text{RANK}(1), \text{RANK}(2), \ldots, \text{RANK}(n)$ exactly in this order. Moreover, after the i-th iteration of the main loop the value $\text{RANK}(i)$ is no longer needed.

In Section 2.1 we have shown that using the recurrence (2) we can generate the sequence $\text{RANK}(1), \text{RANK}(2), \ldots, \text{RANK}(n)$ given the RANKNEXT map and the value $\text{RANK}(1)$. The above observations suggest the algorithm Lcp9 whose code is shown in Fig. 4. In the first step of Lcp9 (Line 1) we call the procedure Sa2RankNext which, for $j = 1, \ldots, n$, stores the value $\text{RANKNEXT}(j)$ in $\text{LCP}[j]$, and returns the value $\text{RANK}(1)$. Then, in the i-th iteration of the main loop (Lines 3–15) given $\text{RANK}(i)$ we retrieve $\text{RANK}(i+1)$ from entry $\text{LCP}[\text{RANK}(i)]$. Note that as soon as we have retrieved $\text{RANK}(i+1)$ we can use the entry $\text{LCP}[\text{RANK}(i)]$ for storing the LCP relative to $\mathtt{t}[i, n]$.

Summing up, the main loop of algorithm Lcp9 (Lines 3–15) works as follows. At the beginning of the i-th iteration the variable k contains the value $\text{RANK}(i)$. In the body of the loop we store in `nextk` the value `lcp[k]` which is $\text{RANK}(i+1)$; then we compute ℓ_i (the LCP between $\mathtt{t}[i, n]$ and the suffix preceding it) and we store it in `lcp[k]`, which is the right place since $\mathtt{k} = \text{RANK}(i)$. Finally, we update k (Line 14) and we start the next iteration. Note that the actual computation of ℓ_i is done as in the Lcp13 algorithm; indeed, lines 5–13 are identical in both algorithms. The only difference between our algorithm and the one of Kasai et al. is the computation of the rank information using the RANKNEXT map rather than the `rank` array.

We conclude observing that the correctness of the procedure Sa2RankNext follows from the correctness of Bwt2RankNext in Fig. 2 and by the relationship

Procedure Lcp9

```
 1. k = Sa2RankNext(lcp);
 2. h=0;
 3. for(i=1;i<=n;i++) {
 4.   nextk = lcp[k];
 5.   if(k==1) lcp[k]=-1;
 6.   else {
 7.     j = sa[k-1];
 8.     while(i+h<=n && j+h<=n
 9.            && t[i+h]==t[j+h])
10.       h++;
11.     lcp[k] = h;
12.   }
13.   if(h>0) h--;
14.   k=nextk;
15. }
```

Procedure Sa2RankNext(rank_next)

```
 1. j = ++count[t[n]];
 2. rank_next[j]=0;
 3. for(i=1;i<=n;i++) {
 4.   if(sa[i]==1)
 5.     eos_pos = i;
 6.   else {
 7.     c = t[sa[i]-1];
 8.     j = ++count[c];
 9.     rank_next[j]=i;
10.   }
11. }
12. return eos_pos;
```

Fig. 4. Algorithm Lcp9 for linear time computation of the LCP array using $9n$ bytes of storage. The algorithm takes as input the text t and the suffix array sa and stores in lcp the LCP array. The procedure Sa2RankNext computes the RankNext map given t and sa. After the procedure call at Line 1 of Lcp9 the RankNext map is stored in the array lcp and the value Rank(1) is stored in the variable k.

between the suffix array and the Burrows-Wheeler Transform (see the procedure Sa2Bwt in Fig. 2).

4 LCP Computation in $(6 + \delta)n$ Bytes of Storage

In this section we describe the algorithm Lcp6 which computes the LCP array "overwriting" the suffix array in the sense that the LCP array is stored in the same array which initially contains the suffix array entries.

Recall that the correctness of the algorithm of Kasai et al. follows from the observation that whenever ℓ_i and ℓ_{i-1} are both defined we have $\ell_i \geq \ell_{i-1} - 1$ (see Section 2.2). The following Lemma (see [12] for the proof) shows that using the Burrows-Wheeler Transform of t we can say something more on the relationship between ℓ_i and ℓ_{i-1}.

Lemma 1. *Let* bwt *denote the Burrows-Wheeler Transform of* t*, and let* $k = \textsc{Rank}(i)$*. If* $k > 1$ *and* $\mathsf{bwt}[k] = \mathsf{bwt}[k-1]$ *then* $\ell_i = \ell_{i-1} - 1$. □

Assume now that the array bwt is available, and consider the main loop of Lcp9 (Lines 3–15 in Fig. 4). At the beginning of the i-th iteration the variable k contains the value $k = \textsc{Rank}(i)$. By Lemma 1, if $\mathsf{bwt}[k] = \mathsf{bwt}[k-1]$ we know that $\ell_i = \ell_{i-1} - 1$. Since ℓ_{i-1} is stored in the variable h, we conclude that, if $\mathsf{bwt}[k] = \mathsf{bwt}[k-1]$, we can skip Lines 8–10 and proceed with the next iteration. This means that for computing the LCP array we only need the values

$\text{SA}[k-1]$'s for all k's such that $\mathsf{bwt}[k] \neq \mathsf{bwt}[k-1]$. This observation is the starting point of our algorithm.

Let z' denote the number of consecutive equal characters in bwt and let $z = n - z'$. In the algorithm Lcp6 (see Figure 5) we evaluate z with a scan of `bwt` and we allocate an array `sa_aux` of size z for storing those suffix array entries that are needed for computing the LCP array (Lines 2–4). Although we already know which suffix array entries must be stored in `sa_aux`, to retrieve these entries efficiently we must store them in the proper order. Let $k_1, k_2, \ldots, k_z$, with $k_1 < k_2 < \cdots < k_z$ denote the indexes such that $\mathsf{bwt}[k_i] \neq \mathsf{bwt}[k_i - 1]$. By the above discussion we know that we must store in `sa_aux` the values $\text{SA}[k_1 - 1], \text{SA}[k_2 - 1], \ldots, \text{SA}[k_z - 1]$. Note, however, that the value $\text{SA}[k_i - 1]$ is needed when we process the suffix $\mathsf{t}[\text{SA}[k_i], n]$. Since the main loop of the LCP algorithm considers the suffixes in the order $\mathsf{t}[1, n], \mathsf{t}[2, n], \ldots, \mathsf{t}[n, n]$ in Lcp6 we store in `sa_aux[i]` the value $\text{SA}[k_{\pi(i)} - 1]$ where π is a permutation of $1, \ldots, z$ such that

$$\text{SA}[k_{\pi(1)}] < \text{SA}[k_{\pi(2)}] < \cdots < \text{SA}[k_{\pi(z)}]. \tag{3}$$

In other words, we store in `sa_aux` the suffix array entries in the order in which they will be used by the LCP algorithm. This will make the retrieval a very simple task.

To obtain such a convenient arrangement of the suffix array entries within `sa_aux`, the algorithm Lcp6 uses the following two-step procedure. In the first step (Lines 6–10) the algorithm computes the RankNext map storing it in the array `sa`. Then, it generates the sequence $\text{Rank}(1), \text{Rank}(2), \ldots, \text{Rank}(n)$ thus traversing the suffix array entries in the order in which they will be considered by the LCP computation. When Lcp6 finds an index k such that $\mathsf{bwt}[k-1] \neq \mathsf{bwt}[k]$ it stores $k-1$ in the next empty position of `sa_aux` (Line 8). Hence, at the end of this first step, for $i = 1, \ldots, z$, the entry `sa_aux[i]` contains the value $k_{\pi(i)} - 1$, where π is the permutation defined by (3). In the second step (Lines 12–14) the algorithm recomputes the suffix array and, with a simple scan over `sa_aux`, stores in `sa_aux[i]` the value $\text{SA}[k_{\pi(i)} - 1]$. Note that we use this elaborate two step procedure simply because we do not want to store at the same time both the suffix array and the RankNext map.

Once the array `sa_aux` is properly initialized, the computation of the LCP array proceeds as in algorithm Lcp9. First, we store the RankNext map in the array `sa` (Line 16). Then, at each iteration of the main loop (Lines 18–30) a RankNext value in `sa` is replaced by a LCP value so that at the end of the loop `sa` contains the LCP array. The computation of the value ℓ_i makes use of Lemma 1. At the beginning of the i-th iteration the variable `k` contains $\text{Rank}(i)$; if `bwt[k-1]==bwt[k]` then ℓ_i is equal to $\ell_{i-1} - 1$ (which is readily available since it is stored in the variable `h`); otherwise we retrieve from `sa_aux` the value $\text{SA}[k-1]$ (Line 23) and we compute ℓ_i with the `while` loop of Lines 24–25.

In our "real world" model the total space occupancy of the above algorithm is $6n + 4z$ bytes: we use $2n$ bytes for the arrays `t` and `bwt`, $4n$ bytes for the array `sa` (which is used for storing the suffix array, the RankNext map, and the LCP array), and $4z$ bytes for `sa_aux`. This latter amount depends on the structure of the input. More precisely, in [10] it is proven that for any $k \geq 1$ we have $z \leq |\Sigma|^k + 2nH_k$, where $|\Sigma|$ is the alphabet size and H_k is the k-th order entropy of the input. In practice, for linguistic texts and other "structured" texts the Burrows-Wheeler Transform usually contains many repetitions and consequently z is relatively small. If $z \approx n/2$ (which is not an unusual value) the total space occupancy of Lcp6 is $\approx 8n$ bytes. In the worst case we have $z = n$ and our algorithm uses $10n$ bytes. However, if at Line 4 we find that $z > 3n/4$—which would yield a space occupancy larger than $9n$ bytes—we can quit Lcp6 and use Lcp9 instead.

Algorithm Lcp6

```
 1. // ----- count how many suffix array entries we need ----
 2. for(z=0,i=2;i<=n;i++)
 3.   if(bwt[i-1]!=bwt[i]) z++;
 4. sa_aux=malloc(z*sizeof(int));               // allocate sa_aux[0,z-1]
 5. // ----- determine order in which suffix array entries are needed----
 6. k = Bwt2RankNext(sa);                       // store RankNext in sa[]
 7. for(v=0,i=2;i<=n;i++) {
 8.   if(bwt[k-1]!=bwt[k])  sa_aux[v++]=k-1;
 9.   k=lcp[k];
10. }
11. // ----- store needed suffix array entries in sa_aux ----
12. RankNext2SuffixArray(sa);                  // store Suffix Array in sa[]
13. for(v=0;v<z;v++)
14.   sa_aux[v] = sa[sa_aux[v]];
15. // ----- compute the lcp array as usual ----
16. k = Bwt2RankNext(sa);                       // store RankNext in sa[]
17. v=h=0;
18. for(i=1;i<=n;i++) {
19.   nextk = sa[k];
20.   if(k==1) sa[k]=-1;
21.   else if(bwt[k-1]==bwt[k]) sa[k]=h;
22.   else {
23.     j = sa_aux[v++];                        // retrieve sa[k-1]
24.     while(i+h<=n && j+h<=n && t[i+h]==t[j+h])
25.       h++;
26.     sa[k] = h;
27.   }
28.   if(h>0) h--;
29.   k=nextk;
30. }
```

Fig. 5. Algorithm Lcp6 for linear time computation of the LCP array using $(6+\delta)n$ bytes of storage. The algorithm takes as input the text `t`, the Burrows-Wheeler Transform `bwt`, and the suffix array `sa` and stores the LCP values in `sa` (thus overwriting the suffix array entries). The algorithm uses an auxiliary array `sa_aux` whose size depends on the structure of the input text. After the procedure calls at Lines 6 and 16 the RankNext map is stored in `sa` and the value Rank(1) is stored in `k`. The procedure call at Line 12 stores the suffix array in the array `sa` overwriting the RankNext map (see comment at the end of Sect. 2.1).

5 Experimental Results

We have tested the algorithms Lcp13, Lcp9, and Lcp6 on a collection of files with different lengths and structures (a more detailed experimental analysis can be found in [12]). For each file we built the suffix array using the ds algorithm [13,14] which is currently one of the fastest suffix array construction algorithm. Then, the text and the suffix array were given as input to the algorithms Lcp13, Lcp9, and Lcp6 and their running times were measured considering (user+system) time averaged over five runs. For all tests we used a 1700 MHz Pentium 4 running GNU/Linux with 1.25GB main memory and 256Kb L2 cache.

In Table 1 we report, for each file and for each algorithm, the running time over file length. For Lcp6 we also report the space occupancy expressed as total space occupancy over file length. The files in Table 1 are ordered by increasing average LCP: a large average LCP indicates that the input file contains many long repeated substrings. Note that the file *etext99.gz* has a very small average LCP since it is a compressed file and essentially consists of a "random" sequence over the alphabet $\{0, 1, \ldots, 255\}$. The file *chr22* is a DNA sequence and consists of an apparently random sequence over the alphabet $\{a, c, g, t\}$: its relatively high average LCP is due to the small cardinality of the underlying alphabet.

Our first observation is that Lcp9 is roughly 10% slower than Lcp13. We also notice that for most files both LCP algorithms are faster than the suffix array construction algorithm. Thus, if we consider the combined time required to compute suffix array and LCP array, the overhead for using Lcp9 is usually less than 5% of the total running time. For the algorithm Lcp6 we observe that it is roughly two times slower that Lcp13. However, we also notice that for most

Table 1. Experimental results for LCP construction algorithms. The second and third column show the size and average LCP of the input file. The fourth column reports the time (microseconds per input byte) for the construction of the suffix array. The next three columns report the time (microseconds per input byte) for the computation of the LCP array using the algorithms Lcp13, Lcp9, and Lcp6 respectively. The last column shows the space used by Lcp6 expressed as total space occupancy over file length.

File	Size (Kb)	Ave. LCP	SA time	Lcp13 time	Lcp9 time	Lcp6 time	Lcp6 space
etext99.gz	38,747	2.65	0.97	1.07	1.18	2.08	9.97
sprot	107,048	89.08	1.49	1.00	1.03	1.90	7.01
rfc	113,693	93.02	1.18	0.89	0.92	1.66	6.86
howto	38,498	267.56	0.99	0.77	0.84	1.48	7.29
reuters	112,022	282.07	2.65	0.91	0.96	1.77	6.58
linux	113,530	479.00	1.04	0.76	0.76	1.35	6.88
jdk13	68,094	678.94	2.67	0.69	0.75	1.33	6.26
etext99	102,809	1,108.63	1.55	1.07	1.10	2.02	7.57
chr22	33,743	1,979.25	0.96	0.92	1.01	1.76	8.34
gcc	84,600	8,603.21	1.87	0.69	0.73	1.30	6.75
w3c	101,759	42,299.75	2.11	0.72	0.79	1.40	6.31

files Lcp6 uses less than $8n$ bytes. The exceptions are, as expected, *calgary.zip* and *chr22*. We conclude that, although Lcp6 is slower than Lcp13 and Lcp9, for most files it yields a significant saving in the peak space occupancy. For very large files the combination of Lcp6 with a "lightweight" suffix sorter [1,14] can be the only way to avoid the (deleterious) use of secondary memory.

References

1. S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 55–69. Springer-Verlag LNCS n. 2676, 2003.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, 2004.
4. W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. of the 44th IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.
5. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, pages 943–955. Springer-Verlag LNCS n. 2719, 2003.
6. 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 Symposium on Combinatorial Pattern Matching (CPM '01)*, pages 181–192. Springer-Verlag LNCS n. 2089, 2001.
7. D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 186–199. Springer-Verlag LNCS n. 2676, 2003.
8. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 200–210. Springer-Verlag LNCS n. 2676, 2003.
9. V. Mäkinen. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003.
10. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. of the 15th Symposium on Combinatorial Pattern Matching*. Springer-Verlag LNCS, 2004.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
12. G. Manzini. Two space saving tricks for linear time LCP computation. Technical Report TR-INF-2004-02-03-UNIPMN, Dipartimento di Informatica, Università Piemonte Orientale, http://www.di.unipmn.it, 2004.
13. G. Manzini and P. Ferragina. Lightweight suffix sorting home page. `http://www.mfn.unipmn.it/~manzini/lightweight`.
14. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 2004. To appear. Preliminary version in Proc. 10th European Symposium on Algorithms (ESA '02).
15. K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '02)*, pages 225–232, 2002.