Permuted Longest-Common-Prefix Array*

Juha Kärkkäinen[1], Giovanni Manzini[2], and Simon J. Puglisi[3]

[1] Department of Computer Science,
University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi
[2] Department of Computer Science,
University of Eastern Piedmont, Italy
manzini@mfn.unipmn.it
[3] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
simon.puglisi@rmit.edu.au

Abstract. The longest-common-prefix (LCP) array is an adjunct to the suffix array that allows many string processing problems to be solved in optimal time and space. Its construction is a bottleneck in practice, taking almost as long as suffix array construction. In this paper, we describe algorithms for constructing the *permuted LCP* (PLCP) array in which the values appear in position order rather than lexicographical order. Using the PLCP array, we can either construct or *simulate* the LCP array. We obtain a family of algorithms including the fastest known LCP construction algorithm and some extremely space efficient algorithms. We also prove a new combinatorial property of the LCP values.

1 Introduction

The suffix array (SA) [13] is a lexicographically sorted list of all the suffixes in a string. The longest-common-prefix (LCP) array stores the lengths of the longest-common-prefixes of adjacent suffixes in SA. Augmenting SA with LCP allows many problems in string processing to be solved in optimal time and space. In particular the LCP array is key for: efficiently simulating traversals of the suffix tree [22,5] (top-down, bottom up, suffix link walks) with the suffix array [1]; pattern matching on the suffix array in attractive theoretical bounds [13,1]; fast disk based suffix array arrangements [3,21]; and compressed suffix trees [4].

Various methods for suffix arrays have been extensively investigated but the LCP array has received much less attention. Several very fast SA construction algorithms have been developed in recent years [17], but there has been no improvement in LCP construction time since the original LCP-from-SA algorithm [8]. Furthermore, the fastest SA construction algorithms are also space economical, which is not the case with LCP construction. This is a problem since space is

* This work is supported by the Academy of Finland grant 118653 (ALGODAN), by the Italy-Israel Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics", and by the Australian Research Council.

G. Kucherov and E. Ukkonen (Eds.): CPM 2009, LNCS 5577, pp. 181–192, 2009.

an even bigger concern than time. For large documents, full representations of the text, the SA, and the LCP array cannot be stored (simultaneously) in RAM. There are many methods for SA addressing this problem, including compressed representations [15], external storage [21], and external and semi-external construction [2,6]. The few LCP related improvements have been concerned with the space too (see Section 2).

In this paper we show that a natural and effective way to reduce the time and space costs of LCP computation is the use of a simple alternative representation of LCP values. Instead of storing them in the classical LCP array we store them in the *permuted LCP* (PLCP) array in which the values appear in position order, rather than lexicographical order. The PLCP array has played a role in previous algorithms implicitly (see Lemma 1) or even explicitly [19,11], but we bring it to the center stage. We use the PLCP array as the central piece that connects a number of techniques (old and new) into a family of algorithms.

One advantage of the PLCP array over the LCP array is that it supports compact representation — in two different ways, in fact: sparse array [11] and succinct bitarray [19]. Each representation can be used for *simulating* the LCP array. The properties of the representations are summarized in Table 1.

Table 1. Properties of PLCP representations. The construction space does not include the space for the text and the SA which are the input to the construction algorithms.

	Space	LCP random access	Construction Time	Construction Space
Full array	n words	$\mathcal{O}(1)$	$\mathcal{O}(n)$	n words
Sparse array	n/q words	$\mathcal{O}(q)$ amortized	$\mathcal{O}(n)$	n/q words
Succinct bitarray	$2n + o(n)$ bits	$\mathcal{O}(1)$	$\mathcal{O}(n)$	n words + $2n$ bits
			$\mathcal{O}(n \log n)$	$3n$ bits

The other advantage of the PLCP array over the LCP array is faster construction. The main contribution of this paper are efficient and space economical construction algorithms. The first one (called Φ in Section 5) is a linear time algorithm that is extremely fast in practice. It constructs the full PLCP array roughly 2.2 times faster than the fastest LCP array construction algorithm. Futhermore, combining the algorithm with an LCP-from-PLCP construction yields the fastest known algorithm for computing the LCP array. More space efficient variants of the algorithm (called Φip and Φx in Section 5) are also faster than other comparable algorithms.

Another contribution of the paper is a novel, non-trivial combinatorial property of the (P)LCP array (Theorem 1), which leads to another PLCP construction algorithm (called IB in Section 5). It enables the construction of the succinct bitarray representation with a peak space usage of only $3n$ bits in addition to the text and the SA. Although the algorithm runs in $\mathcal{O}(n \log n)$ time, it is quite fast in practice.

The speed of the algorithms comes from their mostly sequential access patterns, which enable prefetching and avoid cache misses. In particular, the Φ algorithm accesses only one array non-sequentially in any stage. The sequential access patterns help in reducing space, too. With the exception of Φip, all our construction algorithms process SA in sequential order and can produce the LCP array in sequential order. Thus, as in [18], we can keep SA and LCP on disk without an excessive slow-down obtaining *semi-external* algorithms that need RAM only for the text and the PLCP representation, which is only $3n$ bits for the bitarray and even less for the sparse array.

2 Background and Related Work

Throughout we consider a string $t = t[0..n] = t[0]t[1]\dots t[n]$ of $n+1$ symbols. The first n symbols of t are drawn from a constant ordered alphabet, Σ. The final character $t[n]$ is a special "end of string" character, \$, distinct from and lexicographically smaller than all the other characters in Σ.

For $i = 0, \dots, n$ we write $t[i..n]$ to denote the ***suffix*** of t of length $n-i+1$, that is $t[i..n] = t[i]t[i+1]\cdots t[n]$. For convenience we will frequently refer to suffix $t[i..n]$ simply as "suffix i". Similarly, we write $t[0..i]$ to denote the ***prefix*** of t of length $i+1$. We write $t[i..j]$ to represent the ***substring*** $t[i]t[i+1]\cdots t[j]$ of t that starts at position i and ends at position j.

The ***suffix array*** of t, denoted SA_t or just SA when the context is clear, is an array $\mathrm{SA}[0..n]$ which contains a permutation of the integers $0..n$ such that $t[\mathrm{SA}[0]..n] < t[\mathrm{SA}[1]..n] < \cdots < t[\mathrm{SA}[n]..n]$. In other words, $\mathrm{SA}[j] = i$ iff $t[i..n]$ is the j^{th} suffix of t in ascending lexicographical order.

The ***lcp array*** $\mathrm{LCP} = \mathrm{LCP}[0..n]$ is an array defined by t and SA_t. Let $\mathrm{lcp}(y, z)$ denote the length of the longest common prefix of strings y and z. For every $j \in 1..n$,

$$\mathrm{LCP}[j] = \mathrm{lcp}(t[\mathrm{SA}[j-1]..n], t[\mathrm{SA}[j]..n]),$$

that is, $\mathrm{LCP}[j]$ is the length of the longest common prefix of suffixes $\mathrm{SA}[j-1]$ and $\mathrm{SA}[j]$. $\mathrm{LCP}[0]$ is undefined.

The ***permuted lcp array*** — $\mathrm{PLCP}[0..n-1]$ — has the same contents as LCP but in different order. Specifically, for every $j \in 1..n$,

$$\mathrm{PLCP}[\mathrm{SA}[j]] = \mathrm{LCP}[j]. \tag{1}$$

The LCP array first appeared in the original paper on suffix arrays [13], where Manber and Myers show how to compute the LCP array as a byproduct of their $\mathcal{O}(n \log n)$ time SA construction algorithm. The linear time SA construction algorithm of Kärkkäinen and Sanders can also be modified to produce the LCP array [7]. Other SA construction algorithms could probably be modified so too, but a more attractive approach was introduced by Kasai et al. [8], who gave a simple algorithm to construct the LCP array from an already constructed SA in

$\Theta(n)$ time using $2n$ words of extra space[1]. In the rest of the paper we follow this approach and assume that we compute lcp values knowing the text and SA.

The first improvements to Kasai et al.'s algorithm reduced the extra space to n words by eliminating the need for an extra working array through essentially reordering computation. Two different ways to achieve this have been described by Mäkinen [12] and Manzini [14]. Manzini also gives another algorithm that saves space by overwriting SA with LCP, which is not possible with earlier algorithms. It needs n bytes plus n words of extra space in the worst case but usually significantly less than that.

Recently, Puglisi and Turpin [18] gave another algorithm that can overwrite SA with LCP, using $\mathcal{O}(nv)$ time and $\mathcal{O}(n/\sqrt{v})$ extra space, where v is a parameter that controls a space-time tradeoff. The algorithm accesses SA and produces LCP in a strictly left to right manner allowing SA and LCP to reside on disk without a large penalty in speed. This is the first semi-external LCP construction algorithm. In practice, it needs less than twice the size of the text in primary memory.

The PLCP array first appeared in its succinct bitarray form in [19], where Sadakane introduced it as a concise representation of the LCP array, but he did not address the problem of constructing it. The full or sparse PLCP array is used in the LCP construction algorithm by Khmelev, which has not been published in literature but an implementation is available [11].

3 Storing and Using the PLCP Array

The standard way of storing the PLCP array is an array of integers — PLCP$[0..n-1]$. This ***full array*** representation takes n words of storage and because of (1) supports random access to LCP values in $\mathcal{O}(1)$ time.

Two concise representations of the PLCP array are based on the following key property of the PLCP array.

Lemma 1. *For every* $i \in 1..n-1$, $\text{PLCP}[i] \geq \text{PLCP}[i-1] - 1$. □

The lemma was proven by Kasai et al. [8] in a slightly different form. All efficient (P)LCP construction algorithms (including ours) rely on this property.

The ***sparse PLCP array*** — PLCP_q — of $\lceil n/q \rceil$ integers contains only every q^{th} entry of the PLCP array, i.e., $\text{PLCP}_q[i] = \text{PLCP}[iq]$. Lemma 1 allows us to estimate the missing PLCP entries as follows.

Lemma 2. *For any* $i \in 0..n-1$, *let* $a = \lfloor i/q \rfloor$ *and* $b = i \bmod q$, *i.e.,* $i = aq + b$. *If* $(a+1)q \leq n-1$, *then* $\text{PLCP}_q[a] - b \leq \text{PLCP}[i] \leq \text{PLCP}_q[a+1] + q - b$. *If* $(a+1)q > n-1$, *then* $\text{PLCP}_q[a] - b \leq \text{PLCP}[i] \leq n - i \leq q$. □

Using these bounds, we can compute the actual value of PLCP$[i]$ by doing at most $q + \text{PLCP}_q[a+1] - \text{PLCP}_q[a]$ comparisons (or at most q comparisons if

[1] Throughout we will use "extra space" to mean space in addition to t and SA, including the n words required to hold LCP, unless otherwise stated.

$(a+1)q > n-1$). This requires an access to the text and the SA (or Φ_q, see Section 4). The number of comparisons can be close to n for some i but the average number, over all LCP array entries, is at most $\mathcal{O}(q)$ as shown by the following lemma.

Lemma 3. *Assuming the text and the* SA *are available, the sparse* PLCP *array* PLCP_q *supports random access to* LCP *values in* $\mathcal{O}(q)$ *amortized time.*

Proof. For $1 \leq i \leq n$ it is $\text{LCP}[i] = \text{PLCP}[\text{SA}[i]]$. Hence, if $\text{SA}[i] = qk$ then $\text{LCP}[i] = \text{PLCP}_q[k]$ and computing it takes $\mathcal{O}(1)$ time. Otherwise let $\text{SA}[i] = aq+b$, with $0 \leq b < q$. By Lemma 2 we can compute $\text{LCP}[i]$ comparing at most $q+\text{PLCP}_q[a+1]-\text{PLCP}_q[a]$ symbols of the suffixes $t[\text{SA}[i-1]..n]$ and $t[\text{SA}[i]..n]$ (or at most q symbols if $(a+1)q > n-1$). For $i = 1, \ldots, n$ let $c(i)$ denote the number of comparisons needed for computing $\text{LCP}[i]$. We prove the lemma by showing that

$$\frac{1}{n}\sum_{i=1}^{n} c(i) \leq (q-1) + q^2/n.$$

Let $a' = \lfloor (n-1)/q \rfloor$ so that $n' = a'q$ is the largest multiple of q smaller than n. For the above observation, the indexes i such that $\text{SA}[i] \geq n'$ contribute to the sum $\sum_{i=1}^{n} c(i)$ by at most q^2. To complete the proof we show that

$$\sum_{i:\ SA[i]<n'} c(i) \ \leq\ (q-1)n.$$

Since for $k = 1, \ldots, a'-1$ there are exactly $q-1$ indexes i such that $kq < \text{SA}[i] < (k+1)q$ we have

$$\sum_{i:\ SA[i]<n'} c(i) \ \leq\ (q-1)\sum_{k=0}^{a'-1}(q + \text{PLCP}_q[k+1] - \text{PLCP}_q[k]). \tag{2}$$

Define $w(k) = \text{PLCP}_q[k] + kq$. Then, (2) can be rewritten as

$$\begin{aligned}
\sum_{i:\ SA[i]<n'} c(i) \ &\leq\ (q-1)\sum_{k=0}^{a'-1}(w(k+1) - w(k)) \\
&=\ (q-1)(w(a') - w(0)) \\
&\leq\ (q-1)(\text{PLCP}_q[a'] + a'q) \\
&=\ (q-1)(\text{PLCP}(n') + n') \\
&\leq\ (q-1)n.
\end{aligned}$$

and the lemma follows. □

The ***succinct bitarray*** representation of the PLCP array [19] consists of a bit array $\text{B}[0..2n-1]$, where the $\text{B}[j] = 1$ if and only if $j = 2i + \text{PLCP}[i]$ for some $i \in [0..n-1]$. Note that due to Lemma 1, $2i + \text{PLCP}[i]$ has a different value for

each i. Then $\text{PLCP}[i] = \mathit{select}_1(\text{B}, i+1) - 2i$, where $\mathit{select}_1(\text{B}, j)$ returns the position of the j^{th} 1-bit in B. The select-query can be answered in $\mathcal{O}(1)$ time given an additional data structure of $o(n)$ bits. Several such select-structures are known in the literature, the most practical of these for our purposes is probably the *darray* [16]. Summing up, the succinct bitarray representation takes $2n+o(n)$ bits and supports random access to LCP values in $\mathcal{O}(1)$ time.

We have implemented our own optimized variant of the darray structure using the fact that we do not need support for the rank operation. The data structure consists of the bitvector B (modified as described below) and a sparse PLCP array, PLCP_q. The PLCP_q entries indicate the positions of every q^{th} 1-bit in B, dividing B into n/q blocks (of varying sizes). The select operation finds the position of the j^{th} 1-bit. Locating the correct block is easy with PLCP_q. If the block is small, we simply scan the block bitvector to find the correct bit. A bitvector of $\mathcal{O}(\log n)$ bits can be scanned in $\mathcal{O}(1)$ time using appropriate lookup tables (of size $o(n)$). If the block is large enough, we replace the bitvector for the block with a full PLCP array for that block, i.e., with an array of $q-1$ integers pointing to all the 1-bits within the block. A single lookup is enough to locate the bit we want. The main difference to the darray is that the full PLCP arrays for large blocks are stored over the bitvector B overwriting the bits there.

There are several ways of using the PLCP array depending on the application. For some applications, the PLCP array is just as good as the LCP array. Computing the average LCP value is an example. Most applications, though, need the LCP array. We have two options in this case. First, we can simulate the LCP array using (1). Second, we can compute the LCP array from the PLCP array, which is then discarded.

When turning the PLCP array into the LCP array, there are some space saving techniques worth mentioning. First, it is possible to do this by an in-place permutation, i.e., by using a single array that contains the PLCP array at start and the LCP array at finish. In addition to this array, the in-place permutation needs the SA and n bits used as markers. Second, it is possible to store the LCP array and the SA on disk memory. All our PLCP construction algorithms do just a single sequential scan over the SA. Similarly, constructing the LCP array from the PLCP array can be done in sequential order with respect to the LCP and SA. Thus, in main memory we only need to keep the text and the PLCP array, the latter possibly using a compact representation.

4 Constructing the PLCP Array

4.1 Computing PLCP Using the ϕ Array

Our first PLCP construction algorithm uses another array $\Phi[0..n-1]$.[2] For every $j \in 1..n$,

$$\Phi[\text{SA}[j]] = \text{SA}[j-1].$$

[2] The Φ array is so named because it is in some way symmetric to the Ψ array [20].

```
   — Compute Φ_q
1: for i ← 0 to n − 1 do
2:     if SA[i] mod q = 0 then
3:         Φ[SA[i]/q] ← SA[i−1]
   — Turn Φ_q into PLCP_q
4: ℓ ← 0
5: for i ← 0 to ⌊(n − 1)/q⌋ do
6:     j ← Φ_q[i]
7:     while t[iq+ℓ] = t[j+ℓ] do
8:         ℓ ← ℓ+1
9:     PLCP_q[i] ← ℓ
10:    ℓ ← max (ℓ − q, 0)
```

(a)

```
   — Compute irreducible lcp values
1: for i ← 1 to n do
2:     j ← SA[i − 1]
3:     k ← SA[i]
4:     if t[j − 1] ≠ t[k − 1] then
5:         ℓ ← lcp(t[j..n], t[k..n])
6:         k′ ← ⌈k/q⌉ · q
7:         if PLCP_q[k′] < ℓ−k′+k then
8:             PLCP_q[k′] ← ℓ−k′+k
   — Fill in the other values
9: for i ← 1 to ⌊(n − 1)/q⌋ do
10:    if PLCP_q[i] < PLCP_q[i−1]−q then
11:        PLCP_q[i] ← PLCP_q[i−1]−q
```

(b)

Fig. 1. Algorithms for computing Sparse PLCP with sample rate q. The algorithm in (a) uses Φ; the one in (b) uses Irriducible LCPs and assumes $t[-1] = t[n] = \$$. Setting $q = 1$ produces the Full PLCP representation.

Clearly, the Φ array can be computed easily with a scan of the suffix array. The Φ array is also used in [11], though differently from the way we use it.

The Φ array is closely related to the PLCP array: for every $i \in [0..n-1]$,

$$\text{PLCP}[i] = lcp(i, \Phi[i]).$$

Thus to compute PLCP[i] we just need to compare the suffixes i and $\Phi[i]$, and similar to the algorithm of Kasai et al., Lemma 1 allows us to skip the first PLCP[$i-1$] − 1 characters in the comparison. Also note that we can save space by overwriting Φ with PLCP.

The technique generalizes easily to computing the sparse PLCP array by using the sparse version of the Φ array — Φ_q. The full algorithm is given in Fig. 1(a). We can also easily compute the bitarray B this way. The technique is not very space efficient, though, since we need the full Φ array.

4.2 Computing PLCP Using Irreducible LCPs

Our second technique of computing the PLCP array is based on the concept of ***irreducible*** lcp values. We say that PLCP[i] = lcp($i, \phi[i]$) is *reducible* if $t[i-1] = t[\phi[i]-1]$. Reducible values are easy to compute via the next lemma.

Lemma 4. *If* PLCP[i] *is reducible, then* PLCP[i] = PLCP[$i-1$] − 1.

In essence the same result appeared as Lemma 1 in [14].

The idea of the algorithm is to first compute the irreducible lcp values and then use Lemma 4 to fill in the reducible values. The irreducible lcp values are computed *naively*, i.e., by comparing the suffixes i and $\Phi[i]$ from the beginning. The efficiency of the algorithm is based on the following surprising property of

the irreducible lcp values. The property is a conjecture by Dmitry Khmelev [10], but we provide the first proof for it.

Theorem 1. *The sum of all irreducible lcp values is* $\leq 2n \log n$.

Proof. Let $\ell = \text{PLCP}[i] = lcp(i, j)$ (i.e., $j = \Phi[i]$) be an irreducible lcp value, i.e., $t[i-1] \neq t[j-1]$, $t[i..i+\ell-1] = t[j..j+\ell-1]$ and $t[i+\ell] \neq t[j+\ell]$. For every $k \in 0..\ell-1$, the matching pair of characters $t[i+k] = t[j+k]$ contributes to the lcp value and we account for it as follows.

Consider the suffix tree of the reverse of t, and let v_{i+k} and v_{j+k} be the leafs corresponding to the prefixes $t[0..i+k]$ and $t[0..j+k]$. The nearest common ancestor u of v_{i+k} and v_{j+k} represents the reverse of $t[i..i+k]$ (because $t[i-1] \neq t[j-1]$). If v_{i+k} is in a smaller subtree of u than v_{j+k}, the cost of the pair $t[i+k] = t[j+k]$ is assigned to v_{i+k}, otherwise to v_{j+k}.

Now we show that each leaf v carries a cost of at most $2 \log n$. Whenever v is assigned a cost, this is associated with an ancestor u of v and another leaf w under u. We call u a costly ancestor of v and w a costly cousin of v. We will show that (a) each leaf v has at most $\log n$ costly ancestors, and that (b) for each costly ancestor, there is at most two costly cousins.

To show (a), we use the "smaller half trick". Consider the path from v to the root. At each costly ancestor u, the size of the subtree at least doubles with the addition of the subtree containing w. Thus there are at most $\log n$ costly ancestors. Let v be leaf, u a costly ancestor of v and w a corresponding costly cousin representing the reverse of the strings $t[0..i+k]$, $t[i..i+k]$ and $t[0..j+k]$, respectively. Then either $i = \Phi[j]$ or $j = \Phi[i]$. Suppose the former and assume there is another costly cousin $w' \neq w$ of v with the same costly ancestor. Then w' must represent $t[0..j'+k]$ for $j' = \Phi[i]$. Adding a third costly cousin is then no more possible, which proves (b). □

The theorem is asymptotically tight as shown by the next lemma.

Lemma 5. *For a binary de Bruijn sequence of order* k*, the sum of all irreducible lcp values is* $(k-1)2^{k-1} - \Theta(1)$*. As* $n = 2^k + k - 1$ *is the length of the sequence, the sum of irreducible lcp values is* $(n/2) \log n - \mathcal{O}(n)$.

Proof. Let x be any sequence on $\Sigma = \{0, 1\}$ of length $k-1$. $x0$ and $x1$ both appear in the de Bruijn sequence so they are in contiguous positions of the suffix array. The symbols preceding $x0$ and $x1$ cannot be identical, otherwise the de Bruijn sequence would contain two identical length-k subsequences. Thus, we have an irreducible lcp value $lcp(x0, x1) = k-1$. The lemma follows since there are 2^{k-1} such x's. □

The notion of irreducible lcp can be used to compute all our PLCP representations with the same idea: first compute the irreducible lcp values naively and then fill in the rest using Lemma 4. When computing the full PLCP array the algorithm is trivial and by Theorem 1 requires $\mathcal{O}(n \log n)$ time. With the sparse PLCP array, we use the following fact

$$\text{PLCP}[i] = \max\{\text{PLCP}[j] - (i-j) : j \leq i \text{ and } \text{PLCP}[j] \text{ is irreducible.}\}$$

Table 2. Data files used for empirical tests, sorted in ascending order of average LCP

Name	Mean LCP	Max LCP	Size (bytes)	Description
sprot	89	7,373	109,617,186	Swiss prot database
rfc	93	3,445	116,421,901	RFC text files
linux	479	136,035	116,254,720	Linux kernel 2.4.5 source
jdk13	679	37,334	69,728,899	html/java files from JDK 1.3
etext	1,109	286,352	105,277,340	Gutenberg etext99/*.txt files
chr22	1,979	199,999	34,553,758	Human chromosome 22
gcc	8,603	856,970	86,630,400	gcc 3.0 source files
w3c	42,300	990,053	104,201,579	HTML files from w3c.org

The first stage updates the nearest sparse entry following each irreducible lcp value and the second stage fills in the rest. The full algorithm is in Fig. 1(b).

With the bitarray representation, we need a second bit array C[0..$n-1$] to store the positions of the irreducible entries. For each irreducible entry PLCP[i], we set the bit i in C and the bit PLCP[i] + $2i$ in B. Once done, setting the bits for reducible values is easy. This algorithm needs only $3n$ bits in addition to the text and the suffix array and requires $\mathcal{O}(n \log n)$ time to construct.

5 Experimental Results

For testing we used the files listed in Table 2[3]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 3.4.4) executed with the -O3 option. Times given are the minima of three runs and were recorded with the standard C `getrusage` function.

Experiments measured the time to compute various PLCP representations, and the classical LCP array. All methods tested take as input t and SA, which are not modified. All data structures reside in primary memory. The algorithms and their space requirements are described in Table 3. Included are three previous approaches. For consistency we modified the pt$x$ code to produce LCP in a separate array, not overwriting SA as in [18].

Runtimes for PLCP and LCP array construction are given in Table 4. The runtime of a fast SA construction algorithm[4], is included as a reference. Of the PLCP/LCP algorithms there are several interesting pairings to consider.

Φ vs. klaap. Overall, Φ is clearly the fastest route to the LCP array, being around 1.5 times faster than klaap on all inputs, and roughly 2.2 times faster if one stops at the PLCP. This is no doubt due better locality of memory reference: at any time Φ only access one array in a non-sequential fashion, whereas klaap makes random accesses to two arrays throughout its execution. Φ also consistently shades method I.

[3] Available at `http://web.unipmn.it/~manzini/lightweight/corpus/`
[4] Available at `http://www.michael-maniscalco.com/msufsort.htm`

Table 3. Algorithms and their space requirements. The space requirements do not include arrays that all algorithms share, i.e., text and SA for PLCP construction and text, SA and LCP for LCP construction.

Alg.	PLCP Space	LCP Space	PLCP representation	Description
Φ	n words	n words	full	Φ algorithm
Φx	n/x words	n/x words	sparse ($q = x$)	Φ algorithm
Φip	n words	0	full	Φ using in-place permutation in LCP construction
I	n words	n words	full	Irreducible lcp algorithm
IB	$3n$ bits	$3n$ bits	bitarray	Irreducible lcp algorithm
klaap		n words		Kasai et al [8]
m9		0		Manzini's Lcp9 [14]
pt$x$		$3n/\sqrt{x}$ words		Puglisi and Turpin [18]

Table 4. Runtimes (in milliseconds) for the various LCP construction algorithms

Alg.	sprot	rfc	linux	jdk13	etext	chr22	gcc	w3c	Avg.
Φ PLCP	13.57	14.22	13.12	6.64	15.86	4.89	9.41	9.98	10.96
I PLCP	20.46	20.85	18.64	10.52	23.24	7.91	13.37	15.37	16.30
Φ16 PLCP	3.11	3.58	3.17	1.66	3.41	1.11	2.32	2.47	2.60
Φ32 PLCP	2.45	2.64	2.55	1.45	2.47	0.81	1.88	2.14	2.05
Φ64 PLCP	2.16	2.27	2.23	1.33	2.06	0.67	1.65	1.98	1.79
IB PLCP	17.78	18.40	16.33	6.41	23.91	9.51	10.84	9.68	14.11
Φ LCP	20.55	21.39	19.99	10.62	23.21	6.87	14.24	15.93	16.60
Φip LCP	37.25	39.11	38.31	20.80	37.93	10.88	26.93	32.03	30.41
Φ16 LCP	35.71	41.25	33.01	17.66	43.93	13.94	23.69	25.45	29.33
Φ32 LCP	33.90	37.87	31.36	17.64	36.75	11.77	22.48	25.87	27.21
Φ64 LCP	36.77	37.53	32.07	20.54	34.36	10.55	23.23	29.75	28.10
IB LCP	104.93	108.83	89.27	46.48	123.60	31.22	57.74	63.51	78.20
klaap	34.10	32.70	28.67	16.01	38.29	10.94	21.24	26.23	26.02
m9	54.59	53.86	44.34	27.11	59.76	16.63	32.95	43.02	41.53
pt64	56.50	53.40	46.11	42.45	42.22	11.14	35.03	62.05	43.61
pt256	49.96	47.59	42.10	45.48	41.26	9.96	33.60	67.64	42.20
SA	42.42	38.31	36.09	24.85	44.22	12.83	26.52	35.08	32.54

Φip vs. m9. These algorithms use no extra space for LCP construction. The significantly faster speed of Φip can be largely attributed to an optimization that exploits the out-of-order execution capabilities of the CPU. When performing the in-place permutation, Φip follows multiple chains simultaneously. Execution alternates between the active chains allowing the CPU to proceed with one chain while waiting for a cache miss on another.

Φx and IB vs. pt$x$. These algorithms use very little extra space. Φx is clearly the fastest of these algorithms, nearly as fast as klaap, in fact. On the other hand, IB is quite slow when computing the LCP array due to the slowness of the

select structure. We have not spent much time optimizing the implementation and a significant speed-up may be possible.

Φx, IB and pt$x$ can effectively work with disk resident SA and LCP. Then only the text and the space in Table 3 need to reside in primary memory. We have implemented such *semi-external* versions of the Φx algorithms. Similar to results in [18] we found that the sequential access to SA and LCP of these algorithms meant that runtimes increased by at most 10% (from those in Table 4). For brevity, we do not report actual times here. However, we remark that the semi-external version of $\Phi 64$ constructs the LCP array for a 1Gb prefix of the Human Genome in 526 seconds (ie. under 10 minutes) on our test machine, and allocates, including the space for the text, just 1.06Gb of RAM. For the same file pt64 requires 595 seconds and allocates 2.72Gb.

6 Concluding Remarks

In this paper we have investigated the PLCP array, a simple alternative representation of lcp values in which the values appear in position order, rather than lexicographical order, as they do in the LCP array. The PLCP array is very fast to construct, offers interesting space/time tradeoffs and can be used to simulate the LCP array or efficiently construct a full representation of it.

A drawback of current LCP construction algorithms, including those we describe here, is that they require primary memory at least equal to the size of the input text, so that the random accesses to it do not become expensive disk seeks. An I/O efficent algorithm for LCP array construction is an important direction for future work. I/O efficient algorithms for SA construction are described in [2].

Acknowledgements. Some of the key ideas in this paper were originated by Dmitry Khmelev who tragically died at young age in 2004 [9]. In particular, Theorem 1 is his conjecture [10] and the idea of using the sparse PLCP array comes from his algorithm [11].

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
2. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM Journal of Experimental Algorithmics 12, 1–24 (2008)
3. Ferragina, P., Grossi, R.: The String B-Tree: A new data structure for string search in external memory and its applications. Journal of the ACM 46, 236–280 (1999)
4. Fischer, J., Mäkinen, V., Navarro, G.: An(other) entropy-bounded compressed suffix tree. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 152–165. Springer, Heidelberg (2008)
5. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)

6. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science 387, 249–257 (2007)
7. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
8. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
9. Kelbert, A.: Memorial website of Dima Khmelev (2006), `http://mgg.coas.oregonstate.edu/~anya/dima/index-eng.html`
10. Khmelev, D.: Personal communication (2004)
11. Khmelev, D.: Program lcp version 0.1.9 (2004), `http://www.math.toronto.edu/dkhmelev/PROGS/misc/lcp-eng.html`
12. Mäkinen, V.: Compact suffix array — a space efficient full-text index. Fundamenta Informaticae 56, 191–210 (2003); Special Issue - Computing Patterns in Strings
13. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22, 935–948 (1993)
14. Manzini, G.: Two space saving tricks for linear time LCP computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39 (2007)
16. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX 2007). SIAM, Philadelphia (2007)
17. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39, 1–31 (2007)
18. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for Longest-Common-Prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
19. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 225–232. ACM/SIAM (2002)
20. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 294–313 (2003)
21. Sinha, R., Puglisi, S.J., Moffat, A., Turpin, A.: Improving suffix array locality for fast pattern matching on disk. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 661–672. ACM Press, New York (2008)
22. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th annual Symposium on Foundations of Computer Science, pp. 1–11 (1973)