Sampled Longest Common Prefix Array

Jouni Sirén*

Department of Computer Science, University of Helsinki, Finland
jltsiren@cs.helsinki.fi

Abstract. When augmented with the longest common prefix (LCP) array and some other structures, the suffix array can solve many string processing problems in optimal time and space. A compressed representation of the LCP array is also one of the main building blocks in many compressed suffix tree proposals. In this paper, we describe a new compressed LCP representation: the *sampled LCP array*. We show that when used with a compressed suffix array (CSA), the sampled LCP array often offers better time/space trade-offs than the existing alternatives. We also show how to construct the compressed representations of the LCP array directly from a CSA.

1 Introduction

The suffix tree is one of the most important data structures in string processing and bioinformatics. While it solves many problems efficiently, its usefulness is limited by its size: typically 10–20 times the size of the text [17]. Much work has been put on reducing the size, resulting in data structures such as the enhanced suffix array [1] and several variants of the compressed suffix tree [22, 21, 11, 18].

Most of the proposed solutions are based on three structures: 1) the suffix array, listing the suffixes of the text in lexicographic order; 2) the longest common prefix (LCP) array, listing the lengths of the longest common prefixes of lexicographically adjacent suffixes; and 3) a representation of suffix tree topology. While there exists an extensive literature on compressed suffix arrays (CSA)[1] [19], less has been done on compressing the other structures.

Existing proposals to compress the LCP information are based on the permuted LCP (PLCP) array that arranges the entries in text order. While the PLCP array can be compressed, one requires expensive CSA operations to access LCP values through it. In this paper, we describe the sampled LCP array as an alternative to the PLCP-based approaches. Similar to the suffix array samples used in CSAs, the sampled LCP array often offers better time/space trade-offs than the PLCP-based alternatives.

We also modify a recent PLCP construction algorithm [14] to work directly with a compressed suffix array. Using it, we can construct any PLCP representation with negligible working space in addition to the CSA and the PLCP. A

* Funded by the Academy of Finland under grant 119815.

[1] In this paper, we use the term *compressed suffix array* to refer to any compressed self-index based on the Burrows-Wheeler transform.

variant of the algorithm can also be used to construct the sampled LCP array, but requires more working space. While our algorithm is much slower than the alternatives, it is the first LCP construction algorithm that does not require access to the text and the suffix array. This is especially important for large texts, as the suffix array may not be available or the text might not fit into memory.

We begin with basic definitions and background information in Sect. 2. Section 3 is a summary of previous compressed LCP representations. In Sect. 4, we show how to build the PLCP array directly from a CSA. We describe our sampled LCP array in Sect. 5. Section 6 contains experimental evaluation of our proposals. We finish with conclusions and discussion on future work in Sect. 7.

2 Background

A *string* $S = S[1, n]$ is a *sequence* of *characters* from *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written as $S[i, j]$. A substring of type $S[1, j]$ is called a *prefix*, while a substring of type $S[i, n]$ is called a *suffix*. A *text* string $T = T[1, n]$ is a string terminated by $T[n] = \$ \notin \Sigma$ with lexicographic value 0. The *lexicographic order* "$<$" among strings is defined in the usual way.

The *suffix array (SA)* of text $T[1, n]$ is an array of pointers $\mathsf{SA}[1, n]$ to the suffixes of T in lexicographic order. As an abstract data type, a suffix array is any data structure with similar functionality as the concrete suffix array. This can be defined by an efficient support for the following operations: (a) *count* the number of occurrences of a *pattern* in the text; (b) *locate* these occurrences (or more generally, retrieve a suffix array value); and (c) *display* any substring of T.

Compressed suffix arrays (CSA) [12, 8] support these operations. Their compression is based on the *Burrows-Wheeler transform (BWT)* [3], a permutation of the text related to the SA. The BWT of text T is a sequence $L[1, n]$ such that $L[i] = T[\mathsf{SA}[i] - 1]$, if $\mathsf{SA}[i] > 1$, and $L[i] = T[n] = \$$ otherwise.

The Burrows-Wheeler transform is reversible. The reverse transform is based on a permutation called *LF-mapping* [3, 8]. Let $C[1, \sigma]$ be an array such that $C[c]$ is the number of characters in $\{\$, 1, 2, \ldots, c - 1\}$ occurring in the text. For convenience, we also define $C[0] = 0$ and $C[\sigma + 1] = n$. By using this array and the sequence L, we define LF-mapping as $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in prefix $L[1, i]$.

The inverse of LF-mapping is $\Psi(i) = select_c(L, i - C[c])$, where c is the highest value with $C[c] < i$, and $select_c(L, j)$ is the position of the jth occurrence of character c in L [12]. By its definition, function Ψ is strictly increasing in the range $\Psi_c = [C[c] + 1, C[c + 1]]$ for every $c \in \Sigma$. Additionally, $T[\mathsf{SA}[i]] = c$ and $L[\Psi(i)] = c$ for every $i \in \Psi_c$.

These functions form the backbone of CSAs. As $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ [8] and hence $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$, we can use these functions to move the suffix array position backward and forward in the sequence. Both of the functions can be efficiently implemented by adding some extra information to a compressed representation of the BWT. Standard techniques [19] to support suffix array

operations include *backward searching* [8] for *count*, and adding a sample of suffix array values for *locate* and *display*.

Let $lcp(A, B)$ be the length of the longest common prefix of sequences A and B. The *longest common prefix (LCP) array* of text $T[1, n]$ is the array $\mathsf{LCP}[1, n]$ such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = lcp(T[\mathsf{SA}[i-1], n], T[\mathsf{SA}[i], n])$ for $i > 1$. The array requires $n \log n$ bits of space, and can be constructed in $O(n)$ time [15, 14].

3 Previous Compressed LCP Representations

We can exploit the redundancy in LCP values by reordering them in text order. This results in the *permuted LCP (PLCP) array*, where $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. The following lemma describes a key property of the PLCP array.

Lemma 1 ([15, 14]). *For every* $i \in \{2, \ldots, n\}$, $\mathsf{PLCP}[i] \geq \mathsf{PLCP}[i-1] - 1$.

As the values $\mathsf{PLCP}[i] + 2i$ form a strictly increasing sequence, we can store the array in a bit vector of length $2n$ [22]. Various schemes exist to represent this bit vector in a succinct or compressed form [22, 11, 18].

Space-efficiency can also be achieved by sampling every qth PLCP value, and deriving the missing values when needed [16]. Assume we have sampled $\mathsf{PLCP}[aq]$ and $\mathsf{PLCP}[(a+1)q]$, and we want to determine $\mathsf{PLCP}[aq+b]$ for some $b < q$. Lemma 1 states that $\mathsf{PLCP}[aq] - b \leq \mathsf{PLCP}[aq+b] \leq \mathsf{PLCP}[(a+1)q] + q - b$, so at most $q + \mathsf{PLCP}[(a+1)q] - \mathsf{PLCP}[aq]$ character comparisons are required to determine the missing value. The average number of comparisons over all entries is $O(q)$ [14]. By carefully selecting the sampled positions, we can store the samples in $o(n)$ bits, while requiring only $O(\log^{\delta} n)$ comparisons in the worst case for any $0 < \delta \leq 1$ [10].

Unfortunately these compressed representations are not very suitable for use with CSAs. The reason is that the LCP values are accessed through suffix array values, and *locate* is an expensive operation in CSAs. In addition to that, sampled PLCP arrays require access to the text, using the similarly expensive *display*.

Assume that a CSA has SA sample rate d, and that it computes $\Psi(\cdot)$ in time t_Ψ. To retrieve $\mathsf{SA}[i]$, we compute $i, \Psi(i), \Psi^2(i), \ldots$, until we find a sampled suffix array value. If the sampled value was $\mathsf{SA}[\Psi^k(i)] = j$, then $\mathsf{SA}[i] = j - k$. We find a sample in at most d steps, so the time complexity for *locate* is $O(d \cdot t_\Psi)$. Similarly, to retrieve a substring $T[i, i+l]$, we use the samples to get $\mathsf{SA}^{-1}[d \cdot \lfloor \frac{i}{d} \rfloor]$. Then we iterate the function Ψ until we reach text position $i + l$. This takes at most $d + l$ iterations, making the time complexity for *display* $O((d + l) \cdot t_\Psi)$. From these bounds, we get the PLCP access times shown in Table 1.[2]

Depending on the type of index used, t_ψ varies from $O(1)$ to $O(\log n)$ in the worst case [19], and is close to 1 microsecond for the fastest indexes in practice [7, 18]. This is significant enough that it makes sense to keep t_Ψ in Table 1.

The only (P)LCP representation so far that is especially designed for use with CSAs is Fischer's Wee LCP [10] that is basically the *select* structure from

[2] Some CSAs use LF-mapping instead of Ψ, but similar results apply to them as well.

Table 1. Time/space trade-offs for (P)LCP representations. R is the number of equal letter runs in BWT, q is the PLCP sample rate, and $0 < \delta \leq 1$ is a parameter. The numbers for CSA assume Ψ access time t_Ψ and SA sample rate d.

		Access times	
Representation	**Space (bits)**	**Using SA**	**Using CSA**
LCP	$n \log n$	$O(1)$	$O(1)$
PLCP [22]	$2n + o(n)$	$O(1)$	$O(d \cdot t_\Psi)$
PLCP [11]	$2R \log \frac{n}{R} + O(R) + o(n)$	$O(1)$	$O(d \cdot t_\Psi)$
PLCP [18]	$2R \log \frac{n}{R} + O(R \log \log \frac{n}{R})$	$O(\log \log n)$	$O(d \cdot t_\Psi + \log \log n)$
Sampled PLCP [10]	$o(n)$	$O(\log^\delta n)$	$O((d + \log^\delta n) \cdot t_\Psi)$
Sampled PLCP [16]	$\frac{n}{q} \log n$	$O(q)$	$O((d + q) \cdot t_\Psi)$

Sadakane's bit vector representation [22]. When the bit vector itself would be required to answer a query, some characters of two lexicographically adjacent suffixes are compared to determine the LCP value. This increases the time complexity, while reducing the size significantly. In this paper, we take the other direction by reducing the access time, while achieving similar compression as in the run-length encoded PLCP variants [11, 18].

4 Building the PLCP Array from a CSA

In this section, we adapt the *irreducible LCP algorithm* [14] to compute the PLCP array directly from a CSA.

Definition 1. *For $i > 1$, the* left match *of suffix $T[\mathsf{SA}[i], n]$ is $T[\mathsf{SA}[i-1], n]$.*

Definition 2. *Let $T[j, n]$ be the left match of $T[i, n]$. $\mathsf{PLCP}[i]$ is* reducible*, if $i, j > 1$ and $T[i-1] = T[j-1]$. If $\mathsf{PLCP}[i]$ is not reducible, then it is* irreducible*.*

The following lemma shows why reducible LCP values are called reducible.

Lemma 2 ([14]). *If $\mathsf{PLCP}[i]$ is reducible, then $\mathsf{PLCP}[i] = \mathsf{PLCP}[i-1] - 1$.*

The irreducible LCP algorithm works as follows: 1) find the irreducible PLCP values; 2) compute them naively; and 3) fill in the reducible values by using Lemma 2. As the sum of the irreducible values is at most $2n \log n$, the algorithm works in $O(n \log n)$ time [14].

The original algorithm uses the text and its suffix array that are expensive to access in a CSA. In the following lemma, we show how to find the irreducible values by using the function Ψ instead.

Lemma 3. *Let $T[j, n]$ be the left match of $T[i, n]$. The value $\mathsf{PLCP}[i+1]$ is reducible if and only if $T[i] = T[j]$ and $\Psi(\mathsf{SA}^{-1}[j]) = \Psi(\mathsf{SA}^{-1}[i]) - 1$.*

```
— Compute the PLCP array
1  PLCP[1] ← 0
2  (i, x) ← (1, SA^{-1}[1])
3  while i < n
4      Ψ_c ← rangeContaining(x)
5      if x − 1 ∉ Ψ_c or Ψ(x − 1) ≠ Ψ(x) − 1
6          PLCP[i + 1] ← lcp(Ψ(x))
7      else PLCP[i + 1] ← PLCP[i] − 1
8      (i, x) ← (i + 1, Ψ(x))

— Compute an LCP value
9  def lcp(b)
10     (a, k) ← (b − 1, 0)
11     Ψ_c ← rangeContaining(b)
12     while a ∈ Ψ_c
13         (a, b, k) ← (Ψ(a), Ψ(b), k + 1)
14         Ψ_c ← rangeContaining(b)
15     return k
```

Fig. 1. The irreducible LCP algorithm for using a CSA to compute the PLCP array. Function rangeContaining(x) returns $\Psi_c = [C[c] + 1, C[c + 1]]$ where $x \in \Psi_c$.

Proof. Let $x = \mathsf{SA}^{-1}[i]$. Then $x - 1 = \mathsf{SA}^{-1}[j]$.

"If." Assume that $T[i] = T[j]$ and $\Psi(x - 1) = \Psi(x) - 1$. Then the left match of $T[\mathsf{SA}[\Psi(x)], n] = T[i + 1, n]$ is $T[\mathsf{SA}[\Psi(x - 1)], n] = T[j + 1, n]$. As $i + 1 > 1$ and $j + 1 > 1$, it follows that $\mathsf{PLCP}[i + 1]$ is reducible.

"Only if." Assume that $\mathsf{PLCP}[i + 1]$ is reducible, and let $T[k, n]$ be the left match of $T[i + 1, n]$. Then $k > 1$ and $T[k - 1] = T[i]$. As $T[k - 1, n]$ and $T[i, n]$ begin with the same character, and $T[k, n]$ is the left match of $T[i + 1, n]$, there cannot be any suffix S such that $T[k - 1, n] < S < T[i, n]$. But now $j = k - 1$, and hence $T[i] = T[j]$. Additionally,

$$\Psi(\mathsf{SA}^{-1}[j]) = \Psi(\mathsf{SA}^{-1}[k - 1]) = \mathsf{SA}^{-1}[k] = \mathsf{SA}^{-1}[i + 1] - 1 = \Psi(\mathsf{SA}^{-1}[i]) - 1.$$

The lemma follows. □

The algorithm is given in Fig. 1. We maintain invariant $x = \mathsf{SA}^{-1}[i]$, and scan through the CSA in text order. If the conditions of Lemma 3 do not hold for $T[i, n]$, then $\mathsf{PLCP}[i+1]$ is irreducible, and we have to compute it. Otherwise we reduce $\mathsf{PLCP}[i + 1]$ to $\mathsf{PLCP}[i]$. To compute an irreducible value, we iterate $(\Psi^k(b - 1), \Psi^k(b))$ for $k = 0, 1, 2, \ldots$, until $T[\Psi^k(b - 1)] \neq T[\Psi^k(b)]$. When this happens, we return k as the requested LCP value. As we compute $\Psi(\cdot)$ for a total of $O(n \log n)$ times, we get the following theorem.

Theorem 1. *Given a compressed suffix array for a text of length n, the irreducible LCP algorithm computes the PLCP array in $O(n \log n \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ. The algorithm requires $O(\log n)$ bits of working space in addition to the CSA and the PLCP array.*

We can use the algorithm to build any PLCP representation from Table 1 directly. The time bound is asymptotically tight, as shown in the following lemma.

Lemma 4 (Direct extension of Lemma 5 in [14]). *For an order-k* de Bruijn sequence *on an alphabet of size σ, the sum of all irreducible PLCP values is $n(1 - 1/\sigma) \log_\sigma n - O(n)$.*

The sum of irreducible PLCP values of a random sequence should also be close to $n(1-1/\sigma)\log_\sigma n$. The probability that the characters preceding a suffix and its left match differ, making the PLCP value irreducible, is $(1-1/\sigma)$. On the other hand, the average irreducible value should be close to $\log_\sigma n$ [6]. For a text generated by an order-k Markov source with H bits of entropy, the estimate becomes $n(1-1/\sigma')(\log n)/H$. Here σ' is the effective alphabet size, defined by the probability $1/\sigma'$ that two characters sharing an order-k context are identical.

The following proposition shows that large-scale repetitiveness reduces the sum of the irreducible values, and hence improves the algorithm performance.

Proposition 1. *For a concatenation of r copies of text $T[1,n]$, the sum of irreducible PLCP values is $s+(r-1)n$, where s is the sum of the irreducible PLCP values of T.*

Proof. Let $\mathcal{T} = T_1T_2\cdots T_r$ be the concatenation, $\mathcal{T}_{a,i}$ the suffix starting at $T_a[i]$, and $\mathsf{PLCP}_a[i]$ the corresponding PLCP value. Assume that $T_r[n]$ is lexicographically greater than the other end markers, but otherwise identical to them.

For every i, the suffix array of $\mathcal{T}$ contains a range with values $\mathcal{T}_{1,i}, \mathcal{T}_{2,i}, \ldots, \mathcal{T}_{r,i}$ [18]. Hence for any $a > 1$ and any i, the left match of $\mathcal{T}_{a,i}$ is $\mathcal{T}_{a-1,i}$, making the PLCP values reducible for almost all of the suffixes of T_2 to T_r. The exception is that $\mathcal{T}_{2,1}$ is irreducible, as its left match is $\mathcal{T}_{1,1}$, and hence $\mathsf{PLCP}_2[1] = (r-1)n$.

Let $T[j,n]$ be the left match of $T[i,n]$ in the suffix array of T. Then the left match of $\mathcal{T}_{1,i}$ is $\mathcal{T}_{r,j}$, and $\mathsf{PLCP}_1[i] = \mathsf{PLCP}[i]$. Hence the sum of the irreducible values corresponding to the suffixes of T_1 is s. □

5 Sampled LCP Array

By Lemmas 1 and 2, the local maxima in the PLCP array are among the irreducible values, and the local minima are immediately before them.

Definition 3. *The value $\mathsf{PLCP}[i]$ is* maximal*, if it is irreducible, and* minimal*, if either $i = n$ or $\mathsf{PLCP}[i+1]$ is maximal.*

Lemma 5. *If $\mathsf{PLCP}[i]$ is non-minimal, then $\mathsf{PLCP}[i] = \mathsf{PLCP}[i+1] + 1$.*

Proof. If $\mathsf{PLCP}[i]$ is non-minimal, then $\mathsf{PLCP}[i+1]$ is reducible. The result follows from Lemma 2. □

In the following, R is the number of equal letter runs in BWT.

Lemma 6. *The number of minimal PLCP values is R.*

Proof. Lemma 3 essentially states that $\mathsf{PLCP}[i+1]$ is reducible, if and only if $L[\Psi(\mathsf{SA}^{-1}[i])] = T[i] = T[j] = L[\Psi(\mathsf{SA}^{-1}[j])] = L[\Psi(\mathsf{SA}^{-1}[i]) - 1]$, where $T[j,n]$ is the left match of $T[i,n]$. As this is true for $n-R$ positions i, there are exactly R irreducible values. As every maximal PLCP value can be reduced to the next minimal value, and vice versa, the lemma follows. □

Lemma 7. *The sum of minimal PLCP values is $S - (n - R)$, where S is the sum of maximal values.*

Proof. From Lemmas 5 and 6. □

If we store the minimal PLCP values in SA order, and mark their positions in a bit vector, we can use them in a similar way as the SA samples. If we need $\mathsf{LCP}[i]$, and $\mathsf{LCP}[\Psi^k(i)]$ is a sampled position for the smallest $k \geq 0$, then $\mathsf{LCP}[i] = \mathsf{LCP}[\Psi^k(i)] + k$. As k can be $\Theta(n)$ in the worst case, the time bound is $O(n \cdot t_\Psi)$.

To improve the performance, we sample one out of $d' = n/R^{1-\varepsilon}$ consecutive non-minimal values for some $\varepsilon > 0$. Then there are R minimal samples and at most $R^{1-\varepsilon}$ extra samples. We mark the sampled positions in a bit vector of Raman et al. [20], taking at most $(1 + o(1)) \cdot R \log \frac{n}{R} + O(R) + o(n)$ bits of space. Checking whether an LCP entry has been sampled takes $O(1)$ time.

We use δ codes [5] to encode the actual samples. As the sum of the minimal values is at most $2n \log n$, these samples take at most

$$R \log \frac{2n \log n}{R} + O\left(R \log \log \frac{n}{R}\right) \leq R \log \frac{n}{R} + O(R \log \log n)$$

bits of space. The extra samples require at most $\log n + O(\log \log n)$ bits each. To provide fast access to the samples, we can use dense sampling [9] or directly addressable codes [2]. This increases the size by a factor of $1 + o(1)$, making the total for samples $(1 + o(1)) \cdot R \log \frac{n}{R} + O(R \log \log n) + o(R \log n)$ bits of space.

We find a sampled position in at most $n/R^{1-\varepsilon}$ steps. By combining the size bounds, we get the following theorem.

Theorem 2. *Given a text of length n and a parameter $0 < \varepsilon < 1$, the sampled LCP array requires at most $(2 + o(1)) \cdot R \log \frac{n}{R} + O(R \log \log n) + o(R \log n) + o(n)$ bits of space, where R is the number of equal letter runs in the BWT of the text. When used with a compressed suffix array, retrieving an LCP value takes at most $O((n/R^{1-\varepsilon}) \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ.*

By using the BSD representation [13] for the bit vector, we can remove the $o(n)$ term from the size bound with a slight loss of performance.

When the space is limited, we can afford to sample the LCP array denser than the SA, as SA samples are larger than LCP samples. In addition to the mark in the bit vector, an SA sample requires $2 \log \frac{n}{d}$ bits of space, while an LCP sample takes just $\log v + O(\log \log v)$ bits, where v is the sampled value.

The LCP array can be sampled by a two-pass version of the irreducible LCP algorithm. On the first pass, we scan the CSA in suffix array order to find the minimal samples. Position x is minimal, if x is the smallest value in the corresponding Ψ_c, or if $\Psi(x - 1) \neq \Psi(x) - 1$. As we compress the samples immediately, we only need $O(\log n)$ bits of working space. On the second pass, we scan the CSA in text order, and store the extra samples in an array. Then we sort the array to SA order, and merge it with the minimal samples. As the number of extra samples is $o(R)$, we need $o(R \log n)$ bits of working space.

Theorem 3. *Given a compressed suffix array for a text of length n, the modified irreducible LCP algorithm computes the sampled LCP array in $O(n \log n \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ. The algorithm requires $o(R \log n)$ bits of working space in addition to the CSA and the samples, where R is the number of equal letter runs in the BWT of the text.*

6 Implementation and Experiments

We have implemented the sampled LCP array, a run-length encoded PLCP array, and their construction algorithms as a part of the RLCSA [23].[3] For PLCP, we used the same run-length encoded bit vector as in the RLCSA. For the sampled LCP, we used a gap encoded bit vector to mark the sampled positions, and a stripped-down version of the same vector for storing the samples.

To avoid redundant work, we compute minimal instead of maximal PLCP values, and interleave their computation with the main loop. To save space, we only use *strictly minimal* PLCP values with $\mathsf{PLCP}[i] < \mathsf{PLCP}[i+1] + 1$ as the minimal samples. When sampling the LCP array, we make both of the passes in text order, and store all the samples in an array before compressing them.

For testing, we used a 2.66 GHz Intel Core 2 Duo E6750 system with 4 GB of memory (3.2 GB visible to OS) running a Fedora-based Linux with kernel 2.6.27. The implementation was written in C++, and compiled on g++ version 4.1.2. We used four data sets: human DNA sequences (*dna*) and English language texts (*english*) from the Pizza & Chili Corpus [7], the Finnish language Wikipedia with version history (*fiwiki*) [23], and the genomes of 36 strains of Saccharomyces paradoxus (*yeast*) [18].[4] When the data set was much larger than 400 megabytes, a 400 MB prefix was used instead. Further information on the data sets can be found in Table 2.

Only on the *dna* data set, the sum of the minimal values was close to the entropy-based estimate. On the highly repetitive *fiwiki* and *yeast* data sets, the difference between the estimate and the measurement was very large, as predicted by Proposition 1. Even regular English language texts contained enough large-scale repetitiveness that the sum of the minimal values could not be adequately explained by the entropy of the texts. This suggests that, for many real-world texts, the number of runs in BWT is a better compressibility measure than the empirical entropy.

The sum of minimal PLCP values was a good estimate for PLCP construction time. LCP sampling was somewhat slower because of the second pass. Both algorithms performed reasonably well on the highly repetitive data sets, but were much slower on the regular ones. The overall performance was roughly an order of magnitude worse than for the algorithms using plain text and SA [14].

We measured the performance of the sampled LCP array and the run-length encoded PLCP array on each of the data sets. We also measured the *locate*

[3] The implementation is available at http://www.cs.helsinki.fi/group/suds/rlcsa/.

[4] The yeast genomes were obtained from the Durbin Research Group at the Sanger Institute (`http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp/`).

Table 2. Properties of the data sets. H_5 is the order-5 empirical entropy, σ' the corresponding effective alphabet size, # the number of (strictly) minimal values, and S the sum of those values. $S' = n(1 - 1/\sigma')(\log n)/H_5 - n/\sigma'$ is an entropy-based estimate for the sum of the minimal values. The construction times are in seconds.

		Estimates			Minimal values			Strictly minimal		
Name	**MB**	$\mathbf{H_5}$	σ'	$\mathbf{S'/10^6}$	$\mathbf{\#/10^6}$	$\mathbf{S/10^6}$	**S/n**	$\mathbf{\#/10^6}$	$\mathbf{S/10^6}$	**S/n**
english	400	1.86	2.09	3167	156.35	1736	4.14	99.26	1052	2.51
fiwiki	400	1.09	1.52	3490	1.79	273	0.65	1.17	117	0.28
dna	385	1.90	3.55	4252	243.49	3469	8.59	158.55	2215	5.48
yeast	409	1.87	3.34	4493	15.64	520	1.21	10.05	299	0.70

	Sample rates		PLCP		Sampled LCP	
Name	**SA**	**LCP**	**Time**	**MB/s**	**Time**	**MB/s**
english	8, 16, 32, 64	8, 16	1688	0.24	2104	0.19
fiwiki	64, 128, 256, 512	32, 64, 128	327	1.22	533	0.75
dna	8, 16, 32, 64	8, 16	3475	0.11	3947	0.10
yeast	32, 64, 128, 256	16, 32, 64	576	0.71	890	0.46

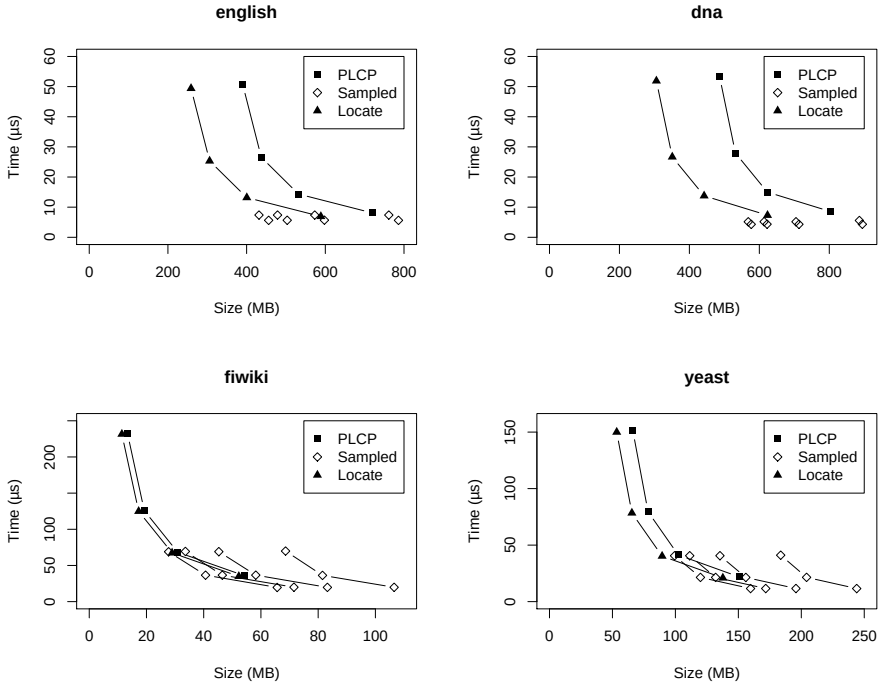

Fig. 2. Time/space trade-offs for retrieving an LCP or SA value. The times are averages over 10^6 random queries. Sampled LCP results are grouped by SA sample rate.

performance of the RLCSA to get a lower bound for the time and space of any PLCP-based approach. The results can be seen in Fig. 2.

The sampled LCP array outperformed PLCP on *english* and *dna*, where most of the queries were resolved through minimal samples. On *fiwiki* and *yeast*, the situation was reversed. As many extra samples were required to get reasonable performance, increasing the size significantly, the sampled LCP array had worse time/space trade-offs than the PLCP array.

While we used RLCSA in the experiments, the results generalize to other types of CSA as well. The reason for this is that, in both PLCP and sampled LCP, the time required for retrieving an LCP value depends mostly on the number of iterations of Ψ required to find a sampled position.

7 Discussion

We have described the sampled LCP array, and shown that it offers better time/space trade-offs than the PLCP-based alternatives, when the number of extra samples required for dense sampling is small. Based on the experiments, it seems that one should use the sampled LCP array for regular texts, and a PLCP-based representation for highly repetitive texts.

In a recent proposal [4], the entire LCP array was compressed by using directly addressable codes (DAC) [2]. The resulting structure is much faster but usually also much larger than the other compressed LCP representations. See the full paper [24] for a comparison between the sampled LCP array and the DAC-based approach.

We have also shown that it is feasible to construct the (P)LCP array directly from a CSA. While the earlier algorithms are much faster, it is now possible to construct the (P)LCP array for larger texts than before, and the performance is still comparable to that of direct CSA construction [23]. On a multi-core system, it is also easy to get extra speed by parallelizing the construction.

It is possible to maintain the (P)LCP array when merging two CSAs. The important observation is that an LCP value can only change, if the left match changes in the merge. An open question is, how much faster the merging is, both in the worst case and in practice, than rebuilding the (P)LCP array.

While the suffix array and the LCP array can be compressed to a space relative to the number of of equal letter runs in BWT, no such representation is known for suffix tree topology. This is the main remaining obstacle in the way to compressed suffix trees optimized for highly repetitive texts.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal on Discrete Algorithms*, 2(1):53–86, 2004.
2. N. R. Brisaboa, S. Ladra, and G. Navarro. Directly adressable variable-length codes. In *SPIRE 2009*, volume 5721 of *LNCS*, pages 122–130. Springer, 2009.

3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
4. R. Cánovas and G. Navarro. Practical compressed suffix trees. Accepted to Symposium on Experimental Algorithms, 2010.
5. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
6. J. Fayolle and M. D. Ward. Analysis of the average depth in a suffix tree under a Markov model. In *Proc. 2005 International Conference on Analysis of Algorithms*, volume AD of *DMTCS Proceedings*, pages 95–104. DMTCS, 2005.
7. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithms*, 13:1.12, 2009.
8. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
9. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *SODA 2007*, pages 690–696. SIAM, 2007.
10. J. Fischer. Wee LCP. arXiv:0910.3123v1 [cs.DS], 2009.
11. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
12. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
13. A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.
14. J. Kärkkäinen, G. Manzini, and S. Puglisi. Permuted longest-common-prefix array. In *CPM 2009*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.
15. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM 2001*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.
16. D. Khmelev. Program lcp version 0.1.9. http://www.math.toronto.edu/dkhmelev/PROGS/misc/lcp-eng.html, 2004.
17. S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.
18. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of individual genomes. In *RECOMB 2009*, volume 5541 of *LNCS*, pages 121–137. Springer, 2009.
19. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.
20. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA 2002*, pages 233–242. SIAM, 2002.
21. L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *LATIN 2008*, LNCS, pages 362–373. Springer, 2008.
22. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
23. J. Sirén. Compressed suffix arrays for massive data. In *SPIRE 2009*, volume 5721 of *LNCS*, pages 63–74. Springer, 2009.
24. J. Sirén. Sampled longest common prefix array. arXiv:1001.2101v2 [cs.DS], 2010.