We prove that longest common prefix (LCP) information can be stored in much less space than previously known. More precisely, we show that in the presence of the text and the suffix array, \( o(n) \) additional bits are sufficient to answer LCP-queries asymptotically in the same time that is needed to retrieve an entry from the suffix array. This yields the smallest compressed suffix tree with sub-logarithmic navigation time.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

Augmenting the suffix-array [7,15] with an additional array holding the lengths of longest common prefixes drastically increases its functionality [15,12]. Stored as a plain array, this so-called LCP-array occupies \( n \lceil \log n \rceil \) bits for a text of length \( n \). Sadakane [21] shows that less space is actually needed, namely \( 2n + o(n) \) bits if the suffix array is available at lookup time (see Section 2.3 for more details). Due to the \( 2n \)-bit term, this data structure is nevertheless incompressible, even for highly regular texts. As a simple example, suppose the text consists of only \( a \)'s. Then the suffix array can be compressed to almost negligible space \([20,9,3]\), while Sadakane’s LCP-array cannot.

Text regularity is usually measured in \( k \)-th order empirical entropy \( H_k \) [16]. We have \( 0 \leq H_k \leq \log \sigma \) for a text on an alphabet of size \( \sigma \), with \( H_k \) being "small" for compressible texts. In Section 3 of this article, we prove that the LCP-array can be stored in \( O(\frac{n}{\log \log n}) \) or \( nH_k + o(n) \) bits (depending on how the text itself is stored), while giving access to an arbitrary LCP-value in time \( O(\log^d n) \) (arbitrary constant \( 0 < d < 1 \)). This should be compared to other compressed or sampled variants of the LCP-array. We are aware of three such methods:

1. Russo et al. [19] achieve \( nH_k + o(n) \) space, but retrieval time at least \( O(\log^{1+\epsilon} n) \), hence super-logarithmic (arbitrary constant \( 0 < \epsilon < 1 \)).
2. Fischer et al. [5] achieve \( nH_k (\log \frac{1}{\log H_k} + O(1)) \) bits, with retrieval time \( O(\log^\beta n) \), again for any constant \( 0 < \beta < 1 \). Although the space vanishes if \( H_k \) does, it is worse than our data structure.
3. Kärkkäinen et al. [12, Lemma 3] also employ the idea of "sampling" the LCP-array, but achieve only amortized time bounds. Allowing the same space as for our data structure \( O(\frac{n}{\log \log n}) \) bits on top of the suffix array and the text), one would have to choose \( q = \log n \log \log n \) in their scheme, yielding super-logarithmic \( O(\log n \log \log n) \) amortized retrieval time.

Finally, in Section 4, we apply our new representation of the LCP-array to suffix trees. This yields the first compressed suffix tree with \( O(nH_k) \) bits of space and sub-logarithmic navigation-time for almost all operations.

2. Definitions and previous results

This section sketches some known data structures that we are going to make use of. Throughout this article, we use the standard word-RAM model of computation, in which we have a computer with word-width \( w \), where
\[ \log n = O(w) \]. Fundamental arithmetic operations (addition, shifts, multiplication, etc.) on \( w \)-bit wide words can be computed in \( O(1) \) time.

2.1. Rank and select on binary strings

Consider a bit-string \( S[1, n] \) of length \( n \). We define the fundamental rank- and select-operations on \( S \) as follows: \( \text{rank}_1(S, i) \) gives the number of 1's in the prefix \( S[1, i] \), and \( \text{select}_1(S, i) \) gives the position of the \( i \)-th 1 in \( S \), reading \( S \) from left to right (\( 1 \leq i \leq n \)). Operations \( \text{rank}_0(S, i) \) and \( \text{select}_0(S, i) \) are defined similarly for 0-bits. There are data structures of size \( O\left(n \log \log n \right) \) bits in addition to \( S \) that support \( O(1) \)-rank- and select-operations, respectively \cite{11, 6}. For an easily accessible exposition of these techniques, we refer the reader to the survey by Navarro and Mäkinen \cite{18, Section 6.1}.

2.2. Suffix- and LCP-arrays

The suffix array \cite{7, 15} for a given text \( T \) of length \( n \) is an array \( A[1, n] \) of integers s.t. \( T[A[i], n] < T[A[i+1], n] \) for all \( 1 \leq i < n \); i.e., \( A \) describes the lexicographic order of \( T \)'s suffixes by "enumerating" them from the lexicographically smallest to the largest. It follows from this definition that \( A \) is a permutation of the numbers \([1, n] \). Take, for example, the string \( T = \text{CACAACCAC} \). Then \( A = [10, 4, 8, 2, 5, 9, 3, 7, 1, 6] \). Note that the suffix array is actually a "left-to-right" (i.e., alphabetical) enumeration of the leaves in the suffix tree \cite[Part II]{10} for \( T \). As \( A \) stores \( n \) integers from the range \([1, n] \), it takes \( n \) words (or \( n \log n \) bits) to store \( A \) in uncompressed form. However, there are also different variants of compressed suffix arrays; see again the survey by Navarro and Mäkinen \cite{18} for an overview of this field. In all cases, the time to access an arbitrary entry \( A[i] \) rises to \( \omega(1) \); we denote this time by \( t_A \). All current compressed suffix arrays have \( t_A = \Omega(\log^2 n) \) in the worst case (arbitrary constant \( 0 < \varepsilon \leq 1 \)), and there are indeed ones that achieve this time \cite{9}.

In the same way as suffix arrays store the leaves of the corresponding suffix tree, the LCP-array captures information on the heights of the internal nodes as follows. Array \( H[1, n] \) is defined such that \( H[i] \) holds the length of the longest common prefix of the lexicographically \((i-1)\)-st and \( i \)-th smallest suffixes. In symbols, \( H[i] = \max[k: \ T[A[i-1..A[i]-1]+k-1] = T[A[i..A[i]+k-1]] \) for all \( 1 < i \leq n \), and \( H[1] = 0 \). For \( T = \text{CACAACCAC} \), \( H = [0, 0, 1, 2, 2, 0, 1, 2, 3, 1] \). Kasai et al. \cite{13} gave an algorithm to compute \( H \) in \( O(n) \) time, and Manzini \cite{17} adapted this algorithm to work in-place.\footnote{Mäkinen \cite[Fig. 3]{14} gives another algorithm to compute \( H \) almost in-place.}

2.3. \( 2n + o(n) \)-Bit representation of the LCP-array

Let us now come to the description of the succinct representation of the LCP-array due to Sadakane \cite{21}. The key to his result is the fact that the LCP-values cannot decrease too much if listed in the order of the inverse suffix array \( A^{-1} \) (defined by \( A^{-1}[i] = j \) iff \( A[j] = i \)), a fact first proved by Kasai et al. \cite[Theorem 1]{13}:

**Proposition 1.** For all \( i > 1 \), \( H[A^{-1}[i]] \geq H[A^{-1}[i-1]] - 1 \).

Because \( H[A^{-1}[i]] \leq n - i + 1 \) (the LCP-value cannot be longer than the length of the suffix!), this implies that \( H[A^{-1}[1]] + 1, H[A^{-1}[2]] + 2, \ldots, H[A^{-1}[n]] + n \) is an increasing sequence of integers in the range \([1, n]\). Now this list can be encoded differentially: for all \( i = 1, 2, \ldots, n \), subsequently write the difference \( d[i] := H[A^{-1}[i]] - H[A^{-1}[i-1]] + 1 \) of neighboring elements in unary code \( 0[d[i]]1 \) into a bit-vector \( S \), where we assume \( H[A^{-1}[1]] = 0 \). Here, \( 0^x \) denotes the juxtaposition of \( x \) zeros. See also Fig. 1. Combining this with the fact that the LCP-values are all less than \( n \), it is obvious that there are at most \( n \) zeros and exactly \( n \) ones in \( S \). Further, if we prepare \( S \) for constant-time range- and select-queries, we can retrieve an entry from \( H \) by

\[
H[i] = \text{rank}_0(S, \text{select}_1(S, A[i])) - A[i].
\]

This is because the select-1-statement gives the position where the encoding for \( H[A[i]] \) ends in \( H \), and the ranko-statements counts the sum of the \( I[j]'s \) for \( 1 \leq j \leq A[i] \). So subtracting the value \( A[i] \), which has been "artificially" added to the LCP-array, yields the correct value. See Fig. 1 for an example. Because \( \text{rank}_0(H, \text{select}(H, x)) = \text{select}_1(H, x) - x \), we can rewrite (1) to

\[
H[i] = \text{select}_1(S, A[i]) - 2A[i].
\]

such that only one select-call has to be invoked. This leads to

**Proposition 2 (Succinct representation of LCP-arrays).** The LCP-array for a text of length \( n \) can be stored in \( 2n + O(\frac{n \log \log n}{\log n}) \) bits in addition to the suffix array, while being able to access its elements in time \( O(t_A) \).

3. Less space, same time

The solution from Section 2.3 is admittedly elegant, but certainly leaves room for further improvements. Because the bit-vector \( S \) stores the LCP-values in text order, we first have to convert the position in \( A \) to the corresponding position in the text. Hence, the lookup time to \( H \) is dominated by \( t_A \), the time needed to retrieve an element from

\[
T = \text{C A C A C A C A C S}\\A = 10 4 8 2 5 9 3 7 1 6\\H = 0 0 1 2 2 0 1 2 3 1\\I = 4 4 4 4 4 7 9 9 9 10\\S = 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 0 1
\]

Fig. 1. Illustration to the succinct representation of the LCP-array.
the compressed suffix array. Intuitively, this means that we could take up to $O(t_A)$ time to answer the select-query, without slowing down the whole lookup asymptotically. Although this is not exactly what we do, keeping this idea in mind is helpful for the proof of the following theorem.

**Theorem 3.** Let $T$ be a text of length $n$ with $O(1)$-access to its characters. Then the LCP-array for $T$ can be stored in $O(n/\log \log n)$ bits in addition to $T$ and to the suffix array, while being able to access its elements in $O(t_A + \log^2 n)$ time (arbitrary constant $0 \leq \delta \leq 1$).

**Proof.** We build on the solution from Section 2.3. Let $j = A[i]$. From (2), we compute $H[i]$ as $select_1(S, i) - 2j$. Computing $A[i]$ takes time $t_A$. Thus, if we could answer the select-query in the same time (using $O(n/\log \log n)$ additional bits), we were done. We now describe a data structure that achieves essentially this. Our description follows in most parts the solution due to Navarro and Mäkinen [18, Section 6.1], except that it does not store sequence $S$ and the lookup-table on the deepest level.

We divide the range of arguments for $select_1$ into subranges of size $\kappa = \lceil \log^2 n \rceil$, and store in $N[j]$ the answer to $select_1(S, \kappa j)$. This table $N[1, \lceil \frac{\kappa}{2} \rceil]$ needs $O(n/\log \log n)$ bits, and divides $S$ into blocks of different size, each containing $\kappa$ 1's (apart from the last).

A block is called long if it spans more than $\kappa^2 = \Theta(\log^4 n)$ positions in $S$, and short otherwise. For the long blocks, we store the answers to all $select_1$-queries explicitly in a table $P$. Because there are at most $\kappa^2$ long blocks, $P$ requires $O(\frac{n}{\log \log n} \times \kappa \times \log \log n) = O(n/\log^4 n \times \log^2 n \times \log \log n) = O(n/\log \log n)$ bits.

Short blocks contain $\kappa$ 1-bits and span at most $\kappa^2$ positions in $S$. We divide again their range of arguments into subranges of size $\lambda = \lfloor \log^2 \kappa \rfloor = \Theta(\log^2 \log \log n)$. In $N'[i]$, we store the answer to $select_1(S, \lambda i)$, this time only relative to the beginning of the block where $i$ occurs. Because the values in $N'$ are in the range $[1, \kappa^2]$, table $N'[1, \lceil \frac{\kappa}{2} \rceil]$ needs $O(\frac{n}{\log \log n})$ bits. Table $N'$ divides the blocks into miniblocks, each containing $\lambda$ 1-bits.

Miniblocks are called long if they span more than $s = \log^2 n$ bits, and short otherwise. For long miniblocks, we store again the answers to all select-queries explicitly in a table $P'$, relative to the beginning of the corresponding block. Because the miniblocks are contained in short blocks of length $\leq \kappa^2$, the answer to such a select-query takes $O(\log \kappa)$ bits of space. Thus, the total space for $P'$ is $O(n/s \times \lambda \times \log \kappa) = O(n/\log \log n)$ bits. This concludes the description of our data structure for select.

To answer a query $select_1(S, i)$, let $a = select_1(S, \lfloor j/\lambda \rfloor \lambda)$ be the beginning of $j$'s mini-block in $S$. Likewise, compute the beginning of the next mini-block as $b = select_1(S, \lfloor j/\lambda \rfloor \lambda + \lambda)$. Now if $b - a > s$, then the mini-block where $i$ occurs is long, and we can look up the answer using our precomputed tables $N$, $N'$, $P$, and $P'$. Otherwise, we return the value $a$ as an approximation to the actual value of $select_1(S, j)$.

We now use the text $T$ to compute $H[i]$ in additional $O(\log \log n)$ time. To this end, let $j' = A[i - 1]$ (see also Fig. 2). The unknown value $H[i]$ equals the length of the longest common prefix of suffixes $T_{j', n}$ and $T_{j', n}$, so we need to compare these suffixes. However, we do not have to compare letters from scratch, because we already know that the first $m = \max(a - 2j, 0)$ characters of $T_{j', n}$ and $T_{j', n}$ match (here with common prefix $\alpha$). At most $s = \log^2 n$ further characters will match (here $\beta$), until reaching a mismatch character $x \approx_{\text{lex}} y$.

If the text is not available for $O(1)$ access, we have two options. First, we can always compress it with recent methods [22,48] to $nH_k + o(n)$ space (which is within the space of all compressed suffix arrays), while still guaranteeing $O(1)$ random access to its characters:

**Corollary 4.** The LCP-array for a text of length $n$ can be stored in $nH_k + O(n/\log \log n + n/\log \log n)$ bits in addition to the suffix array (simultaneously over all $k \in o(\log n)$ for alphabet size $\sigma$), while being able to access its elements in time $O(t_A + \log^2 n)$ (arbitrary constant $0 \leq \delta \leq 1$).

The second option is to use the compressed suffix array itself to retrieve characters in $O(t_A)$ time. This is either already provided by the compressed suffix array [20], or can be simulated [5]. This leads to

**Corollary 5.** Let $A$ be a compressed suffix array for a text of length $n$ with access time $t_A = O(\log^2 n)$. Then the LCP-array can be stored in $o(n)$ bits in addition to the suffix array, while being able to access its elements in time $O(\log^{\epsilon + \delta} n)$ (arbitrary constant $0 < \delta \leq 1$).

Note in particular that all known compressed suffix arrays have worst-case lookup time $\Theta(\log^3 n)$ at the very best, so the requirements on $t_A$ in Corollary 5 are no restriction on its applicability. Further, by choosing $\epsilon$ and $\delta$ such that $\epsilon + \delta < 1$, the time to access the LCP-values remains sub-logarithmic.

### 3.1. Improved retrieval time

Additional time could be saved in Theorem 3 and Corollary 4 by noting that a chunk of $\log_n n$ text characters can be processed in $O(1)$ time in the RAM-model for alphabet size $\sigma$. Hence, when comparing the at most $s = \log^2 n$
characters from suffixes $T_{j+m,n}$ and $T'_{j+m,n}$ (end of the proof of Theorem 3), this could be done by processing at most $s/\log s n = \log \sigma \log^{\delta^{-1}} n$ such chunks. This is especially interesting if the alphabet size is small; in particular, if $\sigma = O(2^{\log^{1-\epsilon} n})$, the retrieval time becomes constant.

The same improvement is possible for Kärkkäinen et al.’s solution [12], resulting in $O(q \log \sigma / \log n)$ amortized retrieval time in their scheme.

4. A small entropy-bounded compressed suffix tree

The data structure from Theorem 3 is particularly appealing in the context of compressed suffix trees. Fischer et al. [5] give a compressed suffix tree that has sub-logarithmic time for almost all navigational operations. It is based on the compressed suffix array due to Grossi et al. [9], a compressed LCP-array, and data structures for range minimum- and previous/next smaller value-queries (RMQ and PNSV). Its size is $n H_k (2 \log \frac{1}{H_k} + \frac{1}{\delta} + O(1)) + o(n)$ bits, where the “ugly” $n H_k (\log \frac{1}{H_k} + O(1))$-term comes from a compressed form of the LCP-array. If we replace this data structure with our new representation, we get (using Corollary 5 for simplicity):

**Theorem 6.** A suffix tree can be stored in $(1 + \frac{1}{\delta}) n H_k + o(n)$ bits such that all operations can be computed in sub-logarithmic time (except level ancestor queries, which have an additional $O(\log n)$ penalty).

**Proof.** The space can be split into $(1 + \frac{1}{\delta}) n H_k + o(n)$ bits from the compressed suffix array [9], additional $o(n)$ bits from the LCP-array of Corollary 5, plus $o(n)$ bits for the RMQ- and PNSV-queries. The time bounds are obtained from the third column of Table 1 in [5].

Other trade-offs than those in Theorem 6 are possible, e.g., by taking different suffix arrays, or by preferring the LCP-array from Corollary 4 over that of Corollary 5.

**Acknowledgements**

The author wishes to express his gratitude towards the anonymous reviewers, whose insightful comments helped to improve the present material substantially.

**References**