Computing All Distinct Squares in Linear Time for Integer Alphabets*

Hideo Bannai[1], Shunsuke Inenaga[2], and Dominik Köppl[3]

1 **Department of Informatics, Kyushu University, Fukuoka, Japan**
bannai@inf.kyushu-u.ac.jp
2 **Department of Informatics, Kyushu University, Fukuoka, Japan**
inenaga@inf.kyushu-u.ac.jp
3 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
dominik.koeppl@tu-dortmund.de

Abstract

Given a string on an integer alphabet, we present an algorithm that computes the set of all distinct squares belonging to this string in time linear in the string length. As an application, we show how to compute the tree topology of the minimal augmented suffix tree in linear time. Asides from that, we elaborate an algorithm computing the longest previous table in a succinct representation using compressed working space.

1998 ACM Subject Classification G.2.1 Combinatorial Algorithms

Keywords and phrases tandem repeats, distinct squares, counting algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.22

1 Introduction

A square is a string of the form SS, where S is some non-empty string. It is well-known that a string of length n contains at most $n^2/4$ squares. This bound is the number of *all* squares, i.e., we count multiple occurrences of the same square, too. If we consider the number of all *distinct* squares, i.e., we count *exactly one* occurrence of each square, then it becomes linear in n: The first linear upper bound was given by Fraenkel and Simpson [17] who proved that a string of length n contains at most $2n$ distinct squares. Later, Ilie [26] showed the slightly improved bound of $2n - \Theta(\lg n)$. Recently, Deza et al. [10] refined this bound to $\lfloor 11n/6 \rfloor$. In the light of these results one may wonder whether future results will "converge" to the upper bound of n: The *distinct square conjecture* [17, 27] is that a string of length n contains at most n distinct squares; this number is known to be independent of the alphabet size [37]. However, there still is a big gap between the best known bound and the conjecture. While studying a combinatorial problem like this, it is natural to think about ways to actually compute the exact number.

This article focuses on a computational problem on distinct squares, namely, we wish to compute (a compact representation of) the set of all distinct squares in a given string. Gusfield and Stoye [23] tackled this problem with an algorithm running in $\mathcal{O}(n\sigma_T)$ time, where σ_T denotes the number of different characters contained in the input text T of length n.

* This work was mainly done while Dominik Köppl visited Kyushu University in Japan under the support by the JSPS Summer Program SP16305. Hideo Bannai and Shunsuke Inenaga were supported in part by JSPS KAKENHI Grant Numbers JP26280003, JP16H02783, JP17H01697.

Although its running time is optimal $\mathcal{O}(n)$ for a constant alphabet, it becomes $\mathcal{O}(n^2)$ for a large alphabet since σ_T can be as large as $\mathcal{O}(n)$.

We present an algorithm (Section 4.1) that computes this set in $\mathcal{O}(n)$ time for a given string of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$. Like Gusfield and Stoye, we can use the computed set to decorate the suffix tree with all squares (Section 5.1). As an application, we provide an algorithm that computes the tree topology of the minimal augmented suffix tree [1] in linear time (Section 5.2). The fastest known algorithm computing this tree topology takes $\mathcal{O}(n \lg n)$ time [5].

For our approach, we additionally need the longest previous factor table [18, 8]. As a side result of independent interest, we show in Section 3 how to store this table in $2n + o(n)$ bits, and give an algorithm that computes it using compressed working space.

2 Definitions

Our computational model is the word RAM model with word size $\Omega(\lg n)$ for some natural number n. Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. An element w in Σ^* is called a ***string***, and $|w|$ denotes its length. We denote the i-th character of w with $w[i]$, for $1 \leq i \leq |w|$. When w is represented by the concatenation of $x, y, z \in \Sigma^*$, i.e., $w = xyz$, then x, y and z are called a ***prefix***, ***substring*** and ***suffix*** of w, respectively. For i, j with $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j in w.

The ***longest common prefix (LCP)*** of two strings is the longest prefix shared by both strings. The ***longest common extension (LCE)*** query asks for the longest common prefix of two suffixes of the *same* string. The time for an LCE query is denoted by t_{LCE}.

A ***factorization*** of a string T is a sequence of non-empty substrings of T such that the concatenations of the substrings is T. Each substring in the factorization is called a ***factor***.

In the rest of this paper, we take a string T of length $n > 0$, and call it ***the text***. We assume that $T[n] = \$$ is a special character that appears nowhere else in T, so that no suffix of T is a prefix of another suffix of T. We further assume that T is read-only; accessing a character costs constant time. We sometimes need the ***reverse*** of T, which is given by the concatenation $T[n-1] \cdots T[1] \cdot T[n] = T[n-1] \cdots T[1]\$$.

The ***suffix tree*** of T is the tree obtained by compacting the trie of all suffixes of T; it has n leaves and at most $n - 1$ internal nodes. The leaf corresponding to the i-th suffix $T[i..n]$ is labeled with i. Each edge e is associated with a non-empty substring x of T called the ***edge label*** of e. Each edge label x is represented by tuple (i, ℓ) of integers such that $T[i..i + \ell - 1] = x$. This way the suffix tree of T takes $\mathcal{O}(n)$ *words* of space, and it can be computed in $\mathcal{O}(n)$ time for strings of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ [11]. The ***string label*** of a node v is defined as the concatenation of all edge labels on the path from the root to v; the ***string depth*** of a node is the length of its string label.

SA and ISA denote the suffix array and the inverse suffix array of T, respectively [36]. The access time to an element of SA is denoted by t_{SA}. LCP is an array such that $\mathsf{LCP}[i]$ is the length of the longest common prefix of $T[\mathsf{SA}[i]..n]$ and $T[\mathsf{SA}[i-1]..n]$ for $i = 2, \ldots, n$. For our convenience, we define $\mathsf{LCP}[1] := 0$. The arrays SA, ISA, and LCP can be constructed in $\mathcal{O}(n)$ time [30, 32, 31].

A ***range minimum query (RMQ)*** asks for the smallest value in a sub-array of an integer array. There are data structures that can answer RMQs on an integer array of length n in constant time while taking $2n + o(n)$ bits of space [15]. An LCE query for the suffixes $T[s..n]$ and $T[t..n]$ can be answered with an RMQ data structure on LCP with the range $[\min(\mathsf{ISA}[s], \mathsf{ISA}[t]) + 1.. \max(\mathsf{ISA}[s], \mathsf{ISA}[t])]$ in constant time.

A ***bit vector*** is a string on the binary alphabet $\{0,1\}$. A ***select query*** on a bit vector asks the position of the i-th '0' or '1' in the bit vector. There is a data structure that can be built in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ bits of working space such that it takes $o(n)$ bits on top of the bit vector, and can answer a select query in constant time [6].

We identify occurrences of substrings with their position and length in the text, i.e., if x is a substring of T, then there is an i with $1 \leq i \leq n$ and an ℓ with $0 \leq \ell \leq n-i+1$ such that $T[i..i+\ell-1] = x$. In the following, we will represent the occurrences of substrings by tuples of position and length. When storing these tuples in a set, we call the set ***distinct***, if there are no two tuples (i,ℓ) and (i',ℓ) such that $T[i..i+\ell-1] = T[i'..i'+\ell-1]$. A special kind of substring is a square: A ***square*** is a string of the form SS for $S \in \Sigma^+$; we call S and $|S|$ the ***root*** and the ***period*** of the square SS, respectively. Like with substrings, we can generate a set containing some occurrences of squares. A set of ***all distinct squares*** is a distinct set of occurrences of squares that is maximal under inclusion.

3 A Compact Representation of the LPF Array

The longest previous factor table LPF of T is formally defined as

$$\mathsf{LPF}[j] := \max\{\ell \mid \text{there exists an } i \in [1..j-1] \text{ such that } T[i..i+\ell-1] = T[j..j+\ell-1]\}.$$

It is useful for computing the ***Lempel-Ziv factorization*** of $T = f_1 \cdots f_z$, which is defined as $f_i = T[k..k+\max(1,\mathsf{LPF}[k])]$ with $k := \sum_{j=1}^{i-1} |f_j| + 1$ for $1 \leq i \leq z$.

In the following, we will use the text $T = \mathtt{ababaaababa\$}$ as our running example whose LPF array is represented by the small numbers above the characters. The Lempel-Ziv factorization of T is given by $\mathtt{a}|\mathtt{b}|\mathtt{aba}|\mathtt{aa}|\mathtt{baba}|\mathtt{\$}$, where the small numbers denote the factor indices, and the vertical bars denote the factor borders.

▶ **Corollary 1.** *Given* LPF*, we can compute the Lempel-Ziv factorization in* $\mathcal{O}(n)$ *time. If the factorization consists of* z *factors, the factorization can be represented by an array of* $z \lg n$ *bits, where the* x*-th entry stores the beginning of the* x*-th factor. Alternatively, it can be represented by a bit vector of length* n *in which we mark the factor beginnings. A select data structure on top of the bit vector can return the length and the position of a factor in constant time.*

Since we will need LPF in Section 4, we are interested in the time and space bounds for computing LPF. We start with the (to the best of our knowledge) state of the art algorithm with respect to time and space requirements.

▶ **Lemma 2** ([9, Theorem 1])**.** *Given* SA *and* LCP*, we can compute* LPF *in* $\mathcal{O}(nt_{\mathsf{SA}})$ *time. Besides the output space of* $n \lg n$ *bits, we only need constant working space.*

Apart from this algorithm, we are only aware of some practical improvements [40, 28].

Let us consider the size of LCP needed in Lemma 2. Sadakane [41] showed a $2n+o(n)$-bits representation of LCP. Thereto he stores the ***permuted longest-common-prefix array*** PLCP defined as $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$ in a bit vector in the following way (also described in [13]): Since $\mathsf{PLCP}[1]+1, \mathsf{PLCP}[2]+2, \ldots, \mathsf{PLCP}[n]+n$ is a non-decreasing sequence with $1 \leq \mathsf{PLCP}[1]+1 \leq \mathsf{PLCP}[n]+n = n$ ($\mathsf{PLCP}[i] \leq n-i$ since the terminal $\$$ is a unique character in T) the values $I[1] := \mathsf{PLCP}[1]$ and $I[i] := \mathsf{PLCP}[i] - \mathsf{PLCP}[i-1] + 1$ ($2 \leq i \leq n$) are non-negative. By writing $I[i]$ in the unary code $0^{I[i]}1$ to a bit vector S subsequently for each $2 \leq i \leq n$, we can compute $\mathsf{PLCP}[i] = \mathrm{select}_1(S,i) - 2i$ and $\mathsf{LCP}[i] = \mathrm{select}_1(S, \mathsf{SA}[i]) - 2\mathsf{SA}[i]$. Moreover, $\sum_{i=1}^{n} I[i] \leq n$ and therefore S is of length at most $2n$.

Table 1 Algorithms computing LPF; space is counted in bits. The output space |LPF| is not considered as working space. $0 < \epsilon \leq 1$ is a constant.

algorithm	time	working space	|LPF|
Lemma 2,[9]	$\mathcal{O}(nt_{\mathsf{SA}})$	$\|\mathsf{SA}\| + \|\mathsf{LCP}\| + \mathcal{O}(\lg n)$	$n \lg n$
Corollary 3,[35, 24]	$\mathcal{O}(n)$	$n \lg n + 2n + \mathcal{O}(\lg n)$	$n \lg n$
Lemma 6,[34]	$\mathcal{O}(n/\epsilon)$	$(1+\epsilon)n \lg n + \mathcal{O}(n)$	$2n + o(n)$
Lemma 6,[16]	$\mathcal{O}(nt_{\mathsf{SA}})$	$\mathcal{O}(n \lg \sigma)$	$2n + o(n)$

By using Sadakane's LCP-representation, we get LPF with the algorithm of Crochemore et al. [9] in the following time and space bounds:

▶ **Corollary 3.** *Having* SA *and* LCP *stored in* $n \lg n$ *bits (this allows* $t_{\mathsf{SA}} = \mathcal{O}(1)$*) and* $2n + o(n)$ *bits, respectively, we can compute* LPF *with* $\mathcal{O}(\lg n)$ *additional bits of working space (not counting the space for* LPF*) in* $\mathcal{O}(n)$ *time.*

By plugging in a suffix array construction algorithm like the in-place construction algorithm by Goto [21], we get the bounds shown in Table 1.

Although this result seems compelling, this approach stores SA and LPF in plain arrays (the former for getting constant time access). In the following, we will show that the LPF array can be stored more compactly. We start with a new representation of LPF, for which we use the same trick as for PLCP due to the following property (which is crucial for squeezing PLCP into $2n + o(n)$ bits).

▶ **Lemma 4.** $n - j \geq \mathsf{LPF}[j] \geq \mathsf{LPF}[j-1] - 1$ *for* $2 \leq j \leq n$.

Proof. There is an i with $1 \leq i < j - 1$ such that $T[i..i + \mathsf{LPF}[j-1] - 1] = T[j-1..j-1 + \mathsf{LPF}[j-1] - 1]$. Hence $T[i+1..i + \mathsf{LPF}[j-1] - 1] = T[j..j-1 + \mathsf{LPF}[j-1] - 1]$. ◀

We conclude that the sequence $\mathsf{LPF}[1] + 1, \mathsf{LPF}[2] + 2, \ldots, \mathsf{LPF}[n] + n$ is non-decreasing with $1 \leq \mathsf{LPF}[1] + 1 \leq \mathsf{LPF}[n] + n \leq n$. We immediately get:

▶ **Corollary 5.** LPF *can be represented by a bit vector with a select data structure such that accessing an* LPF *value can be performed in constant time. The data structures use* $2n + o(n)$ *bits.*

To get a better working space bound, we have to come up with a new algorithm since the algorithm of Lemma 2 creates a plain array to get constant time random write-access for computing the entries of LPF. To this end, we present two algorithms that compute LPF in this representation with the aid of the suffix tree. The two algorithms are derivatives of the algorithms [34, 16] that compute the Lempel-Ziv factorization, either in $\mathcal{O}(n \lg \lg \sigma)$ time using $\mathcal{O}(n \lg \sigma)$ bits, or in $\mathcal{O}(n/\epsilon^2)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits, for a constant $0 < \epsilon \leq 1$. The current bottleneck of both algorithms is the suffix tree implementation with respect to space and time. Due to current achievements [39, 35], the algorithms now run in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits, or in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits, respectively.

We aim at building the LPF-representation of Corollary 5 directly such that we do not need to allocate the plain LPF array using $n \lg n$ bits in the first place. To this end we create a bit vector of length $2n$ and store the LPF values in it successively. In more detail, we follow the description of the Lempel-Ziv factorization algorithms presented in [34, 16]. There, the algorithms are divided into several passes. In each pass we successively visit leaves in text

order (determined by the labels of the leaves). To compute LPF, we only have to do a single pass. Similarly to the first passes of the two Lempel-Ziv algorithms, we use a bit vector B_V to mark already visited internal nodes. On visiting a leaf we climb up the tree until reaching the root or an already marked node. In the former case (we climbed up to the root) we output zero. In the latter case, we output the string depth of the marked node. By doing so, we have computed LPF$[1..j]$ after having processed the leaf with label j.

▶ **Lemma 6.** *We can compute* LPF *in* $\mathcal{O}(nt_{\mathsf{SA}})$ *time with* $\mathcal{O}(n \lg \sigma)$ *bits of working space, or in* $\mathcal{O}(n/\epsilon)$ *time using* $(1+\epsilon)n \lg n + \mathcal{O}(n)$ *bits of working space, for a constant* $0 < \epsilon \leq 1$. *Both variants include the space of the output in their working spaces.*

Proof. Computing the string depth of a node needs access to an RMQ data structure of LCP, and an access to SA. Both accesses can be emulated by the compressed suffix array in t_{SA} time, given that we have computed PLCP in the above representation. ◀

4 The Set of All Distinct Squares

Given a string T, our goal is to compute all distinct squares of T. Thereto we return a set of pairs, where each pair (s, ℓ) consists of a starting position s and a length ℓ such that $T[s..s+\ell-1]$ is the leftmost occurrence of a square. The size of this set is linear due to

▶ **Lemma 7** (Fraenkel and Simpson [17])**.** *A string of length* n *can contain at most* $2n$ *distinct squares.*

We follow the approach of Gusfield and Stoye [23]. Their idea is to compute a set of squares (the set stores pairs of position and length like described in Section 2)[1] with which they can generate all distinct squares. They call this set of squares a ***leftmost covering set***. A leftmost covering set obeys the property that every square of the text can be constructed by right-rotating a square of this set. A square (k, ℓ) is constructed by ***right-rotating*** a square (i, ℓ) with $i \leq k$ iff each tuple $(i+j, \ell)$ with $1 \leq j \leq k-i$ represents a square $T[i+j..i+\ell+j-1] = T[i+j..i+\ell-1]T[i..i+j-1]$.

The set of the leftmost occurrences of all squares is a set of all distinct squares. Unfortunately, the leftmost covering set computed in [23] is not necessarily a set of all distinct squares since (a) it does not have to be distinct, and (b) a square might be missing that can be constructed by right-rotating a square of the computed leftmost covering set.

For illustration, the squares of our running example $T = \overline{\overline{\texttt{ababa}}\texttt{aa}}\,\underline{\underline{\texttt{ababa}}}\texttt{\$}$ are highlighted with bars. The set of all squares is $\{(1,4),(2,4),(5,2),(6,2),(7,4),(8,4)\}$. If we take the leftmost occurrences of all squares, we get $\{(1,4),(2,4),(5,2)\}$; this set comprises all squares marked by the solid bars, i.e., the dotted bars correspond to occurrences of squares that are not leftmost. In this example, the dotted bars form the set $\{(6,2),(7,4),(8,4)\}$, which is a set of all distinct squares. A leftmost covering set is $\{(1,4),(5,2)\}$.

Our goal is to compute the set of all leftmost occurrences directly by modifying the algorithm of [23]. To this end, we briefly review how their approach works: They compute their leftmost covering set by examining the borders between all Lempel-Ziv factors $f_1 \cdots f_z = T$. That is because of

[1] It differs to the set we want to compute by the fact that they allow, among others, occurrences of the same square in their set.

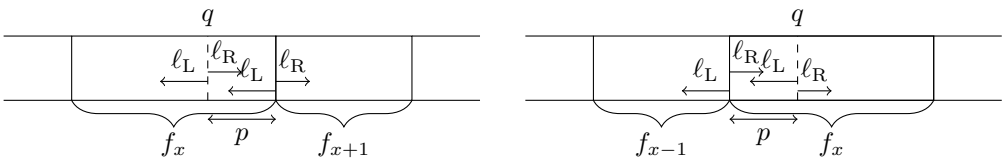


Figure 1 Search for squares on Lempel-Ziv borders. The left image corresponds to squares of type Lemma 8(1), the right image to the type Lemma 8(2). Given two adjacent factors, we determine a position q that is p positions away from the border (the direction is determined by the type of square we want to search for). By two LCE queries we can determine the lengths ℓ_L and ℓ_R that indicate the presence of a square if $\ell_L + \ell_R \geq p$.

▶ **Lemma 8** ([23, Theorem 5]). *The leftmost occurrence of a square $T[i..i+2p-1]$ touches at least two Lempel-Ziv factors. Let f_x $(1 \leq x \leq z)$ be the factor that contains the center of the square $i+p-1$. Then either*

(a) *the square has its left end (position i) inside f_x and its right end (position $i+2p-1$) inside f_{x+1}, or*

(b) *the left end of the square extends into f_{x-1} (or even further left). The right end can be contained inside f_x or f_{x+1}.*

Having a data structure for computing LCE queries on the text and on its inverse, they can probe at the borders of two consecutive factors whether there is a square. Roughly speaking, they have to check at most $|f_x| + |f_{x+1}|$ many periods at the borders of every two consecutive factors f_x and f_{x+1} due to the above lemma ($1 \leq x \leq z$, set f_{z+1} to the empty string). This gives $\sum_{x=1}^{z} t_{\mathrm{LCE}}\,(|f_x| + |f_{x+1}|) = \mathcal{O}(nt_{\mathrm{LCE}})$ time, during which they can compute a leftmost covering set L. Figure 1 visualizes how the checks are done. Applying the algorithm on our running example will yield the set $L = \{(1,4),(5,2),(7,4)\}$. To transform this set into a set of all distinct squares, their algorithm runs the so-called Phase II that uses the suffix tree. It begins with computing the locations of the squares belonging to a subset $L' \subseteq L$ in the suffix tree in $\mathcal{O}(n)$ time. This subset L' is still guaranteed to be a leftmost covering set. Finally, their algorithm computes all distinct squares of the text by right-rotating the squares in L'. In their algorithm, the right-rotations are done by *suffix link walks* over the suffix tree. Their running time analysis is based on the fact that each node has at most σ_T incoming suffix links, where σ_T denotes the number of different characters occurring in the text T. Given that the number of distinct squares is linear, Phase II runs in $\mathcal{O}(n\sigma_T)$ time.

4.1 Algorithm Computing the Set of All Distinct Squares

In the following, we will present our modification of the above sketched algorithm. To speed up the computation, we discard the idea of using the suffix links for right-rotating squares (i.e., we skip Phase II completely). Instead, we compute a list of all distinct squares directly. To this end, we show a modification of the sketched algorithm such that it outputs this list sorted first by the lengths (of the squares), and second by the starting position.

First, we want to show that we can change the original algorithm to output its leftmost covering set in the above described order. To this end, we iterate over all possible periods, and search not yet reported squares at all Lempel-Ziv borders, for each period. To achieve linear running time, we want to skip a factor f_x when the period becomes longer than $|f_x| + |f_{x+1}|$. We can do this with an array Z of $z \lg z$ bits that is zero initialized. When the currently tested period p exceeds $|f_x| + |f_{x+1}|$, we write $Z[x] \leftarrow \min\{y > x : |f_y| + |f_{y+1}| \geq p\}$ such

that $Z[x]$ refers to the next factor whose length is sufficiently large. By doing so, if $Z[x] \neq 0$, we can skip all factors f_y with $y \in [x..Z[x]-1]$ in constant time. This allows us running the modified algorithm still in linear time.

We have to show that the modified algorithm still computes the same set. To this end, let us fix the period p (over which we iterate in the outer loop). By [23, Lemma 7], processing squares satisfying Lemma 8(1) before processing squares satisfying Lemma 8(2) (all squares have the same period p) produces the desired output for period p.

Finally, we show the modification that computes all distinct squares (instead of the original leftmost covering set). On a high level, we use an RMQ data structure on LPF to filter already found squares. The filtered squares are used to determine the leftmost occurrences of all squares by right-rotation. In more detail, we modify Algorithm 1 of [23] by filtering the squares in the following way (see Algorithm 1 in the full version [2]): For each period p, we use a bit vector B marking the beginning positions of all found squares with period p. On reporting a square, we additionally mark its starting position in B. By doing so, an invariant of the algorithm below is that all right-rotated squares of a marked square are already reported.

Let us assume that we are searching for the leftmost occurrences of all squares whose periods are equal to p. Given the starting position s of a square returned by [23, Algorithm 1], we consider the square $(s, 2p)$ and its right-rotations as candidates of our list: If $B[s] = 1$, then this square and its right-rotations have already been reported. Otherwise, we report $(s, 2p)$ if $\mathsf{LPF}[s] < 2p$. In order to find the leftmost occurrences of all not yet reported right-rotated squares efficiently, we first compute the rightmost position e of the repetition of period p containing the square $(s, 2p)$ by an LCE query. Second, we check the interval $I := [s+1..\min(s+p-1, e-2p+1)]$ for the starting positions of the squares whose LPF values are less than $2p$. To this end, we perform an RMQ query on LPF to find the position j whose LPF value is minimal in I. If $\mathsf{LPF}[j] > 2p$, then there is no leftmost occurrence of a square with the period p in the considered range. Otherwise, we report $(j, 2p)$ and recursively search for the text position with the minimal LPF value within the intervals $[s+1..j-1]$ and $[j+1..\min(s+p-1, e-2p+1)]$. In overall, the time of the recursion is bounded by twice the number of distinct squares starting in the interval I, since a recursion step terminates if it could not report any square.

▶ **Theorem 9.** *Given an LCE data structure with t_{LCE} access time and LPF, we can compute all distinct squares in $\mathcal{O}(nt_{LCE} + \mathrm{occ}) = \mathcal{O}(nt_{LCE})$ time, where* occ *is the number of distinct squares.*

Proof. We show that the returned list is the list of all distinct squares. No square occurs in the list twice since we only report the occurrence of a square (i, ℓ) if $\mathsf{LPF}[i] < \ell$. Assume that there is a square missing in the list; let (i, ℓ) be its leftmost occurrence. There is a square (j, ℓ) reported by the (original) algorithm [23] such that $i - \ell/2 < j \leq i$ and right-rotating (j, ℓ) yields (i, ℓ). Since we right-rotate all found squares, we obviously have reported (j, ℓ).

The occ term in the running time is dominated by the nt_{LCE} term due to Lemma 7. ◀

The next corollary, which is immediate from Theorem 9, yields the main result.

▶ **Corollary 10.** *Given a string T of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$, we can compute all distinct squares in T in $\mathcal{O}(n)$ time.*

4.2 Need for RMQ on LPF

Our algorithm performs right-rotations of a square $(s, 2p)$ with an RMQ on the interval $I := [s+1..\min(s+p-1, e-2p+1)]$, where e is the last position of the maximal repetition of period p that contains the square. Without an RMQ data structure, we could linearly scan all LPF values in I, giving $\mathcal{O}(p) = \mathcal{O}(n)$ time. We cannot do better since the LPF values are arbitrary in general. For instance, consider the text $T =$ `abaaabaababaaabaaa$`. The text aligned with LPF is shown in the table below.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T	a	b	a	a	a	b	a	a	b	a	b	a	a	a	b	a	a	a	$
LPF	0	0	1	2	4	3	4	3	2	8	7	6	5	5	4	3	2	1	0

The square `abaaabaa` has two occurrences starting at positions 1 and 10. The square `baaabaaa` at position 11 is found by right-rotating the occurrence of `abaaabaa` at position 10. It is found by a linear scan over LPF or an RMQ on LPF. A slight modification of this example can change the LPF values around this occurrence. This shows that we cannot perform a shortcut in general (like stopping the search when the LPF value is at least twice as large as p).

4.3 Practical Evaluation

We have implemented the algorithm computing the leftmost occurrences of all squares in C++11 [33]. The primary focus was on the execution time, rather than on a small memory footprint: We have deliberately chosen plain 32-bit integer arrays for storing all array data structures like SA, LCP and LPF. These data structures are constructed as follows: First, we generate SA with `divsufsort` [38]. Subsequently, we generate LCP with the Φ-algorithm [29], and LPF with the simple algorithm of [9, Proposition 1]. Finally, we use the bit vector class and the RMQ data structure provided by the sdsl-lite library [20]. In practice, it makes sense to use an RMQ only for very large LCP values and periods (i.e., RMQs on LPF) due to its long execution time. For small values, we naively compared characters, or scanned LPF linearly.

We ran the algorithm on all 200MiB collections of the Pizza&Chili Corpus [12]. The Pizza&Chili Corpus is divided in a real text corpus with the prefix PC, and in a repetitive corpus with the prefix PCR. The experiments were conducted on a machine with 32 GB of RAM and an Intel® Xeon® CPU `E3-1271 v3`. The operating system was a 64-bit version of Ubuntu Linux 14.04 with the kernel version 3.13. We used a single execution thread for the experiments. The source code was compiled using the GNU compiler `g++ 6.2.0` with the compile flags `-O3 -march=native -DNDEBUG`.

Table 2 shows the running times of the algorithm on the described datasets. It seems that large factors tend to slow down the computation, since the algorithm has to check all periods up to $\max_x(|f_x| + |f_{x+1}|)$. This seems to have more impact on the running time than the number of Lempel-Ziv factors z.

4.4 Online Variant

In this section, we consider the *online* setting, where new characters are appended to the end of the text T. Given the text $T[1..i]$ up to position i with the Lempel-Ziv factorization $f_1 \cdots f_y = T[1..i]$, we consider computing the set of all distinct squares of $f_1 \cdots f_{y-2}$, i.e., up to the last two Lempel-Ziv factors. For this setting, we show that we can compute the set of all distinct

Table 2 Practical evaluation of the algorithm computing all distinct squares on the datasets described in Section 4.3. Execution time is in seconds, K = 10^3. It is the median of several conducted experiments, whose variance in time was small. The expression $\text{avg}_{\mathsf{LCP}}$ is the average of all LCP values, and z is the number of Lempel-Ziv factors.

collection	σ	$\text{avg}_{\mathsf{LCP}}$	z	$\max_x \lvert f_x \rvert$	$\max_x \lvert f_x f_{x+1} \rvert$	$\lvert\text{occ}\rvert$	time
PC-DBLP.XML	97	44	7035K	1K	1K	7K	70
PC-DNA	17	60	13,970K	98K	98K	133K	310
PC-ENGLISH	226	9390	13,971K	988K	1094K	13K	2639
PC-PROTEINS	26	278	20,875K	46K	68K	3108K	245
PC-SOURCES	231	373	11,542K	308K	308K	340K	792
PCR-CERE	6	3541	1447K	176K	185K	47K	535
PCR-EINSTEIN.EN	125	45,983	50K	907K	1634K	18,193K	3953
PCR-KERNEL	161	149,872	775K	2756K	2756K	9K	6608
PCR-PARA	6	2268	1927K	71K	74K	37K	265

squares in $\mathcal{O}\left(n \min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ time using $\mathcal{O}(n)$ words of space. To this end, we adapt the algorithm of Theorem 9 to the online setting. We need an algorithm computing LPF online, and a semi-dynamic LCE data structure (answering LCE queries on the text *and* on the reversed text while supporting appending characters to the text).

The main idea of our solution is to build suffix trees with two online suffix tree construction algorithms. The first is Ukkonen's algorithm that computes the suffix tree online in $\mathcal{O}(nt_{\text{nav}})$ time [43], where t_{nav} is the time for inserting a node and navigating (in particular, selecting the child on the edge starting with a specific character). We can adapt this algorithm to compute LPF online: Assume that we have computed the suffix tree of $T[1..i-1]$. The algorithm processes the new character $T[i]$ by (1) taking the suffix links of the current suffix tree, and (2) adding new leaves where a branching occurs. On adding a new leaf with suffix number i, we additionally set $\mathsf{LPF}[i]$ to the string depth of its parent. By doing so, we can update the LPF values in time linear in the update time of the suffix tree. We build the semi-dynamic RMQ data structure of Fischer [14] (or of [42] if n is known beforehand) on top of LPF. This data structure takes $\mathcal{O}(n)$ words and can perform query and appending operations in constant amortized time.

The second suffix tree construction algorithm is a modified version [4] of Weiner's algorithm [44] that builds the suffix tree in the reversed order of Ukkonen's algorithm in $\mathcal{O}(nt_{\text{nav}})$ time. Since Weiner's algorithm incrementally constructs the suffix tree of a given text from right to left, we can adapt this algorithm to compute the suffix tree of the reversed text online in $\mathcal{O}(nt_{\text{nav}})$ time.

To get a suffix tree construction time of $\mathcal{O}\left(n \min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$, we use the predecessor data structure of Beame and Fich [3]. We create a predecessor data structure to store the children of each suffix tree node, such that we get the navigation time $t_{\text{nav}} = \mathcal{O}\left(\min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ for both suffix trees. We also create a predecessor data structure to store the out-going suffix link of each node of the suffix tree constructed by Weiner's algorithm. Overall, these take a total of $\mathcal{O}(n)$ words of space.

Finally, our last ingredient is a dynamic lowest common ancestor data structure with $\mathcal{O}(n)$ words that performs querying and modification operations in constant time [7]. The lowest common ancestor of two suffix tree leaves with the labels j and k is the node whose string depth is equal to the longest common extension of $T[j..i]$ and $T[k..i]$ — remember that we consider the text T up to the position i, hence $T[j..i]$ is (currently) the j-th suffix. Building this data structure on the suffix tree of the text T and on the suffix tree of the reversed text allows us to compute LCE queries in both directions in constant time.

Given the text $T[1..i] = f_1 \cdots f_y$ up to the i-th character, the entries of

$\mathsf{LPF}[1..|f_1 \cdots f_{y-2}| - 1]$ are fixed (i.e., they will not change when appending new characters) due to the properties of the Lempel-Ziv factorization. We let the semi-dynamic RMQ data structure grow with LPF, but only up to the fixed range of LPF. Similarly, the text positions from 1 up to $|f_1 \cdots f_{y-2}| - 1$ are represented as leaves in both suffix trees that are fixed, i.e., these leaves will always be leaves representing their respective suffixes. To sum up, our data structures support LCE queries and RMQs on LPF in the range $[1..|f_1 \cdots f_{y-2}| - 1]$ in constant time.

We adapt the algorithm of Section 4.1 by switching the order of the loops (again). The algorithm first fixes a Lempel-Ziv factor f_x and then searches for squares with a period between one and $|f_x| + |f_{x+1}|$. Unfortunately, we would need an extra bit vector for each period so that we can track all found leftmost occurrences. Instead, we use the predecessor data structure of [3] storing the found occurrences of squares as pairs of starting positions and lengths. These pairs can be stored in lexicographic order (first sorted by starting position, then by length). The predecessor data structure will contain at most occ elements, hence takes $\mathcal{O}(\text{occ}) = \mathcal{O}(n)$ words of space. An insertion or a search costs us $\mathcal{O}\Big(\min\Big(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\Big)\Big)$ time.

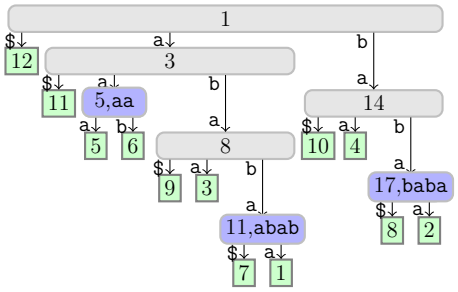
Let us assume that we have computed the set for $T[1..i-1]$, and that the Lempel-Ziv factorization of $T[1..i-1]$ is $f_1 \cdots f_y$. If appending a new character $T[i]$ will result in a new factor f_{y+1}, we check for squares of type Lemma 8(1) and Lemma 8(2) at the borders of f_{y-1}. Duplicates are filtered by the predecessor data structure storing all already reported leftmost occurrences. The algorithm outputs only the leftmost occurrences with the aid of LPF, whose entries are fixed up to the last two factors (this is sufficient since we search for the starting position of the leftmost occurrence of a square with type Lemma 8(1) only in $T[1..|f_1 \cdots f_{y-1}|]$, including right-rotations). In overall, we need $\mathcal{O}\Big((|f_{y-1}| + |f_y|)\min\Big(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\Big)\Big)$ time.

5 Applications

In this section, we provide two applications of the (offline) variant.

5.1 Decorating the Suffix Tree with All Squares

Gusfield and Stoye described a representation of the set of all distinct squares by a decoration of the suffix tree, like the highlighted nodes (additionally annotated with its respective square) shown in the suffix tree of our running example below.



This representation asks for a set of tuples of the form (node, length) such that each square $T[i..i+\ell-1]$ is represented by a tuple (v, ℓ), where v is the highest node whose string label has $T[i..i+\ell-1]$ as a (not necessarily proper) prefix. We show that we can compute

this set of tuples in linear time by applying the Phase II algorithm [23] described in Section 4 to our computed set of all distinct squares. The Phase II algorithm takes a list L_i storing squares starting at text position i, for each $1 \leq i \leq n$. Each of these lists has to be sorted in descending order with respect to the squares' lengths. It is easy to adapt our algorithm to produce these lists: On reporting a square (i, ℓ), we insert it at the front of L_i. By doing so, we can fill the lists *without* sorting, since we iterate over the period length in the outer loop, while we iterate over all Lempel-Ziv factors in the inner loop.

Finally, we can conduct Phase II. In the original version, the goal of Phase II was to decorate the suffix tree with the endpoints of a subset of the original leftmost covering set. We will show that performing exactly the same operations with the set of the leftmost occurrences of all squares will decorate the suffix tree with all squares directly. In more detail, we first augment the suffix tree leaf having label i with the list L_i, for each $1 \leq i \leq n$. Subsequently, we follow Gusfield and Stoye [23] by processing every node of the suffix tree with a bottom-up traversal. During this traversal we propagate the lists of squares from the leaves up to the root: An internal node u inherits the list of the child whose subtree contains the leaf with the smallest label among all leaves in the subtree rooted at u. If the edge to the parent node contains the ending position of one or more squares in the list (these candidates are stored at the front of the list), we decorate the edge with these squares, and pop them off from the list. By [23, Theorem 8], there is no square of the set L' (defined in Section 4) neglected during the bottom-top traversal. The same holds if we exchange L' with our computed set of all distinct squares:

▶ **Lemma 11.** *By feeding the algorithm of Phase II with the above constructed lists L_i containing the leftmost occurrences of the squares starting at the text position i, it will decorate the suffix tree with all distinct squares.*

Proof. We adapt the algorithm of Section 4.1 to build the lists L_i. These lists contain the leftmost occurrences of all squares. In the following we show that no square is left out during the bottom-up traversal. Let us take a suffix tree node u with its children v and w. Without loss of generality, assume that the smallest label among all leaves contained in the subtree of v is smaller than the label of every leaf contained in w's subtree. For the sake of contradiction, assume that the list of w contains the occurrence of a square (i, ℓ) at the time when we pass the list of v to its parent u. The length ℓ is smaller than v's string depth, otherwise it would already have been popped off from the list. But since v's subtree contains a leaf whose label j is the smallest among all labels contained in the subtree of w, the square occurs before at $T[j..j+\ell-1] = T[i..i+\ell-1]$, a contradiction to the distinctness. ◀

This concludes the correctness of the modified algorithm. We immediately get:

▶ **Theorem 12.** *Given* LPF*, an LCE data structure on the reversed text, and the suffix tree of T, we can decorate the suffix tree with all squares of the text in $\mathcal{O}(nt_{\mathrm{LCE}})$ time. Asides from these data structures, we use $(\mathrm{occ}+n)\lg n + z\lg z + \min(n+o(n), z\lg n) + \mathcal{O}(\lg n)$ bits of additional working space.*

▶ **Corollary 13.** *We can compute the suffix tree and decorate it with all squares of the text in $\mathcal{O}(n/\epsilon)$ time using $(3n + \mathrm{occ} + 2n\epsilon)\lg n + z\lg z + \mathcal{O}(n)$ bits, for a constant $0 < \epsilon \leq 1$.*

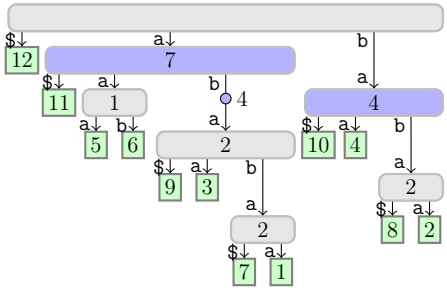
As an application, we consider the common squares problem: Given a set of non-empty strings with a total length n, we want to find all squares that occur in every string in $\mathcal{O}(n)$ time. We solve this problem by first decorating the generalized suffix tree built on all strings with the distinct squares of all strings. Subsequently, we apply the $\mathcal{O}(n)$ time solution of

Hui [25] that annotates each internal suffix tree node v with the number of strings that contain v's string label. This solves our problem since we can simply report all squares corresponding to nodes whose string labels are found in all strings. This also solves the problem asking for the longest common square of all strings in $\mathcal{O}(n)$ time, analogously to the longest common substring problem [22].

The last subsection is dedicated to another application of our suffix tree decoration:

5.2 Computing the Tree Topology of the MAST in Linear Time

A modification of the suffix tree is the ***minimal augmented suffix tree (MAST)*** [1]. This tree can answer the number of the non-overlapping occurrences of a substring S of T in $\mathcal{O}(|S|)$ time. The MAST can be built in $\mathcal{O}(n \lg n)$ time [5].



In this section, we show how to compute the tree topology of the MAST in linear time. The topology of the MAST differs to the suffix tree topology by the fact that the root of each square is the string label of an MAST node. Our goal is to compute a list storing the information about where to insert the missing nodes. The list stores tuples consisting of a node v and a length ℓ; we use this information later to create a new node w splitting the edge (u, v) into (u, w) and (w, v), where u is the (former) parent of v. We will label (w, v) with the last ℓ characters and (u, v) with the rest of the characters of the edge label of (u, v).

To this end, we explore the suffix tree with a top-down traversal while locating the roots of the squares in the order of their lengths. To locate the roots of the squares in linear time we use two data structures. The first one is a semi-dynamic lowest marked ancestor data structure [19]. It allows marking a node and querying for the lowest marked ancestor of a node in constant amortized time. We will use it to mark the area in the suffix tree that has already been processed for finding the roots of the squares.

The second data structure is the list of tuples of the form (node, length) computed in Section 5.1, where each tuple (v, ℓ) consists of the length ℓ of a square $T[i..i + \ell - 1]$ and the highest suffix tree node v whose string label has $T[i..i + \ell - 1]$ as a (not necessarily proper) prefix. We sort this list, which we now call L, with respect to the square lengths with a linear time integer sorting algorithm.

Finally, we explain the algorithm locating the roots of all squares. We successively process all tuples of L, starting with the shortest square length. Given a tuple of L containing the node v and the length ℓ, we want to split an edge on the path from the root to v and insert a new node whose string depth is $\ell/2$. To this end, we compute the lowest marked ancestor u of v. If u's string depth is smaller than $\ell/2$, we mark all descendants of u whose string depths are smaller than $\ell/2$, and additionally the children of those nodes (this can be done by a DFS or a BFS). If we query for the lowest marked ancestor of u again, we get an ancestor w whose string depth is at least $\ell/2$, and whose parent has a string depth less than $\ell/2$. We report w and the subtraction of $\ell/2$ from w's string depth (if $\ell/2$ is equal to the string depth

of w, then w's string label is equal to the root of v's string label, i.e., we do not have to report it).

▶ **Theorem 14.** *We can compute the tree topology of the MAST in linear time using linear number of words.*

Proof. By using the semi-dynamic lowest marked ancestor data structure, we visit a node as many times as we have to insert nodes on the edge to its parent, plus one. This gives $\mathcal{O}(n + 2\mathrm{occ}) = \mathcal{O}(n)$ time. ◀

Open Problems. It is left open to compute the number of the non-overlapping occurrences of the string labels of the MAST nodes in linear time. Since RMQ data structures are practically slow, we wonder whether we can avoid the use of any RMQ without loosing linear running time. The current bottleneck of the online algorithm is the predecessor data structure in terms of the running time. Future integer dictionary data structures can improve the overall performance of this algorithm.

Acknowledgements. We thank Thomas Schwentick for the question whether we can run our algorithm online, for which we provided a solution in Section 4.4.

References

1 Alberto Apostolico and Franco P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996. doi:10.1007/BF01955046.

2 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets, 2016. arXiv:1610.03421.

3 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. doi:10.1006/jcss.2002.1822.

4 Anselm Blumer, Janet A. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985. doi:10.1016/0304-3975(85)90157-4.

5 Gerth Stølting Brodal, Rune B. Lyngsø, Anna Östlin, and Christian N. S. Pedersen. Solving the string statistics problem in time $\mathcal{O}(n \log n)$. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002)*, volume 2380 of *LNCS*, pages 728–739. Springer, 2002. doi:10.1007/3-540-45465-9_62.

6 David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: http://hdl.handle.net/10012/64.

7 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4), 2005. doi:10.1137/S0097539700370539.

8 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008. doi:10.1016/j.ipl.2007.10.006.

9 Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. LPF computation revisited. In Jirí Fiala, Jan Kratochvíl, and Mirka Miller, editors, *Proceedings of the 20th International Workshop on Combinatorial Algorithms (IWOCA 2009)*, volume 5874 of *LNCS*, pages 158–169. Springer, 2009. doi:10.1007/978-3-642-10217-2_18.

10 Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Appl. Math.*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

12 Paolo Ferragina and Gonzalo Navarro. The Pizza & Chili Corpus. Available at http://pizzachili.di.unipi.it and http://pizzachili.dcc.uchile.cl, 2005.

13 Johannes Fischer. Wee LCP. *Inf. Process. Lett.*, 110(8–9):317–320, 2010. doi:10.1016/j.ipl.2010.02.010.

14 Johannes Fischer. Inducing the LCP-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS 2011)*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011. doi:10.1007/978-3-642-22300-6_32.

15 Johannes Fischer and Volker Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

16 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel-Ziv computation in small space (LZ-CISS). In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015. doi:10.1007/978-3-319-19929-0_15.

17 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

18 Frantisek Franek, Jan Holub, William F. Smyth, and Xiangdong Xiao. Computing quasi suffix arrays. *J. Autom. Lang. Comb.*, 8(4):593–606, 2003.

19 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

20 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

21 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets, 2017. arXiv:1703.01009.

22 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

23 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004. doi:10.1016/j.jcss.2004.03.004.

24 Wing-Kai Hon and Kunihiko Sadakane. Space-economical algorithms for finding maximal unique matches. In Alberto Apostolico and Masayuki Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002. doi:10.1007/3-540-45452-7_13.

25 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM 1992)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6_19.

26 Lucian Ilie. A note on the number of squares in a word. *Theor. Comput. Sci.*, 380(3):373–376, 2007. doi:10.1016/j.tcs.2007.03.025.

27 Natasa Jonoska, Florin Manea, and Shinnosuke Seki. A stronger square conjecture on binary words. In Viliam Geffert, Bart Preneel, Branislav Rovan, Julius Stuller, and A Min

Tjoa, editors, *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *LNCS*, pages 339–350. Springer, 2014. doi:10.1007/978-3-319-04298-5_30.

28 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

29 Juha Kärkkäinen, Giovanni Manzini, and Simon John Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.

30 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

31 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.

32 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/j.jda.2004.08.002.

33 Dominik Köppl. Computing all distinct squares efficiently, 2017. URL: https://github.com/koeppl/distinct_squares.

34 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 3–12. IEEE Computer Society, 2016. doi:10.1109/DCC.2016.38.

35 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting, 2016. arXiv:1610.08305.

36 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

37 Florin Manea and Shinnosuke Seki. Square-density increasing mappings. In Florin Manea and Dirk Nowotka, editors, *Proceedings of the 10th International Conference on Combinatorics on Words (WORDS 2015)*, volume 9304 of *LNCS*, pages 160–169. Springer, 2015. doi:10.1007/978-3-319-23660-5_14.

38 Yuta Mori. libdivsufsort, 2015. URL: https://github.com/y-256/libdivsufsort.

39 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In Philip N. Klein, editor, *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

40 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. doi:10.1007/978-3-642-21458-5_4.

41 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

42 Yohei Ueki, Diptarama, Masatoshi Kurihara, Yoshiaki Matsuoka, Kazuyuki Narisawa, Ryo Yoshinaka, Hideo Bannai, Shunsuke Inenaga, and Ayumi Shinohara. Longest common subsequence in at least k length order-isomorphic substrings. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *Proceedings of the 43rd International Conference on Current Trends in Theory and Practice*

of Computer Science (SOFSEM 2017), volume 10139 of *LNCS*, pages 363–374. Springer, 2017. doi:10.1007/978-3-319-51963-0_28.

43 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

44 Peter Weiner. Linear pattern matching algorithms. In H. Raymond Strong, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

Small Observation. In [23, Line 6 of Algorithm 1b], the condition $start + k < h_1$ has to be changed to $start + k \leq h_1$. Otherwise, given the text $T =$ abaabab\$, the algorithm would find only the square aa, but not abaaba.

A Algorithm Execution with one Step at a Time

In this section, we process the running example $T =$ ababaaababa\$ with the algorithm devised in Section 4.1 step by step. SA, LCP, PLCP, and LPF are given in the table below (the LZ row partitions the text into factors, their borders are represented by the vertical bars):

i	1	2	3	4	5	6	7	8	9	10	11	12
T	a	b	a	b	a	a	a	b	a	b	a	\$
SA	12	11	5	6	9	3	7	1	10	4	8	2
LCP	0	0	1	2	1	3	3	5	0	2	2	4
PLCP	5	4	3	2	1	2	3	2	1	0	0	0
LPF	0	0	3	2	1	2	5	4	3	2	1	0
LZ	f_1 |	f_2 |		f_3		|	f_4	|		f_5		| f_6 |

The text $T = \overset{1}{\mathsf{a}}|\overset{2}{\mathsf{b}}|\mathsf{aba}|\overset{4}{\mathsf{aa}}|\mathsf{baba}|\overset{6}{\mathsf{\$}} = f_1 \cdots f_6$ is factorized in six Lempel-Ziv factors. We call $T[1+|f_1 \cdots f_{i-1}|]$ (first position of the i-th factor) and $T[1+|f_1 \cdots f_i|]$ (position after the i-th factor) the ***left border*** and the ***right border*** of f_i, respectively. The idea of the algorithm is to check the presence of a square at a factor border and at an offset value q of the border with LCE queries. q is either the *addition* of p to the *left* border, or the *subtraction* of p from the *right* border (see Figure 1).

The algorithm finds the leftmost occurrences of all squares in the order (first) of their lengths and (second) of their starting positions. We start with the period $p = 1$ and try to detect squares at each Lempel-Ziv factor border. To this end, we create a bit vector B marking all found squares with period $p = 1$. A square of this period is found at the right border of f_3. It is of type Lemma 8(1), since its starting position is in f_3. To find it, we take the right border $b = 6$ of f_3, and the position $q := b - p = 5$. We perform an LCE query at b and q in the forward and backward direction. Only the forward query returns the non-zero value of one. But this is sufficient to find the square aa of period one. Its LPF value is smaller than $2p = 2$, so it is the leftmost occurrence. It is not yet marked in B, thus we have not yet reported it. Right-rotations are not necessary for period 1. Having found all squares with period 1, we clear B.

Next, we search for squares with period 2. We find a square of type Lemma 8(2) at the left border $b = 2$ of f_2. To this end, we perform an LCE query starting from b and $q := b + p = 4$ in both directions. Both LCE queries show that $T[1..5]$ is a repetition with period $p = 2$. Thus we know that $T[1..4]$ is a square. It is not yet marked in B, and has an LPF value smaller than $2p = 4$, i.e., it is a not yet reported leftmost occurrence. On finding a leftmost occurrence of a square, we right-rotate it, and report all right-rotations whose LPF values are below $2p$. This is the case for $T[2..5]$, which is the leftmost occurrence of the square `baba`.

After some unsuccessful checks at the next factor borders, we come to factor f_5 and search for a square of type Lemma 8(2). Two LCE queries in both directions at the left border $b = 8$ of f_5 and $q := b + p = 10$ reveal that $T[7..11]$ is a repetition of period 2. The substring $T[7..10]$ is a square, but its LPF value is $5 (\geq 2p)$, i.e., we have already reported this square. Although we have already reported it, some right-rotation of it might not have been reported yet (see Section 4.2 for an example). This time, all right-rotations (i.e., $T[8..12]$) have an LPF value $\geq 2p$, i.e., there is no leftmost occurrence of a square of period 2 found by right-rotations. In overall, we have found and reported the leftmost occurrences of all squares *once*.

B More Evaluation

Table 3 Running times in seconds, evaluated on different input sizes. We took prefixes of 1MiB, 10MiB, 50MiB, and 100MiB of all collections.

collection	1MiB	10MiB	50MiB	100MiB	200MiB
PC-DBLP.XML	0.2	3	16	33	70
PC-DNA	0.3	3	23	56	310
PC-ENGLISH	0.2	5	42	500	2639
PC-PROTEINS	0.3	4	25	74	245
PC-SOURCES	0.2	3	31	286	792
PCR-CERE	0.6	6	30	79	535
PCR-EINSTEIN.EN	0.4	12	83	1419	3953
PCR-KERNEL	0.2	8	233	1274	6608
PCR-PARA	0.4	4	26	98	265

C Proofs

Proof of Theorem 12

Proof. We need $(\mathrm{occ} + n) \lg n$ bits for storing the lists L_i ($\mathrm{occ} \lg n$ bits for storing the lengths of all squares in an integer array, and $n \lg n$ bits for the pointers to the first element of each list). The array Z uses $z \lg z$ bits. The Lempel-Ziv factors are represented as in Corollary 1. The time t_{LCE} is the maximum time of the LCE data structure and the suffix tree for answering an LCE query. ◀

Proof of Theorem 13

Proof. We use Theorem 6 to store SA, ISA, LCP, and LPF in $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits. Subsequently, we build an RMQ data structure on LCP such that LCE queries can be

answered in constant time. We additionally need the suffix array, its inverse, and the LCP array (with an RMQ data structure) of the reversed text to answer LCE queries on the reversed text. Finally, we endow LPF with an RMQ data structure for the right-rotations. An LCE query on the text can be answered by the string depth of a lowest common ancestor in the suffix tree in constant time. ◀

D Pseudo Code

Algorithm 1: Modified Algorithm 1 of [23].

```
1  b(f) denotes the left end of a factor f = T[b(f)..b(f)+|f|-1], lcp and lcs compute the LCE
     in T and the LCE in the reverse of T (mirroring the input indices by i ↦ n-i for
     1 ≤ i ≤ n-1), respectively.
2  Let f_1, ..., f_z be the factors of the Lempel-Ziv factorization
3  f_{z+1} ← T[n]                                                    // dummy factor
4  Function recursive-rotate(s : starting position, e: ending position)
5      m ← LPF.RMQ[s..e]
6      if m > 2p then return
7      report(m, 2p) and B[m] ← 1
8      recursive-rotate(s,m-1) and recursive-rotate(m+1,e)
9  Function right-rotate(s : starting position of square, p: period of square)
10     if B[s] = 1 then return
11     if LPF[s] < 2p then report(s, 2p) and B[s] ← 1
12     ℓ ← lcp(s, s+p)
13     recursive-rotate(s+1, s+p-1, s+ℓ-p)
14 Z ← array of size z lg z bits, zero initialized
15 m ← max(|f_1|+|f_2|, ..., |f_{z-1}|+|f_z|)
16 for p = 1, ..., m do
17     B ← bit vector of length n, zero initialized
18     for x = 1, ..., z do
19         if |f_x| + |f_{x+1}| < p then
20             y ← x
21             while |f_y| + |f_{y+1}| < p do
22                 if Z[y] ≠ 0 then y ← Z[y]
23                 else incr y
24             Z[x] ← y and x ← y
25         if |f_x| ≥ p then // probe for squares satisfying Lemma 8(1)
26             q ← b(f_{x+1}) - p
27             ℓ_R ← lcp(b(f_{x+1}), q) and ℓ_L ← lcs(b(f_{x+1}) - 1, q - 1)
28             if ℓ_R + ℓ_L ≥ p and ℓ_R > 0 then // found a square of length 2p with its
                right end in f_{x+1}
29                 s ← max(q - ℓ_L, q - p + 1) // square starts at s
30                 right-rotate(s, p)
31         q ← b(f_x) + p // probe for squares satisfying Lemma 8(2)
32         ℓ_R ← lcp(b(f_x), q) and ℓ_L ← lcs(b(f_x) - 1, q - 1)
33         s ← max(b(f_x) - ℓ_L, b(f_x) - p + 1) // square starts in a factor preceding f_x
34         if ℓ_R + ℓ_L ≥ p and ℓ_R > 0 and s + p ≤ b(f_{x+1}) and ℓ_L > 0 then // found a square
            of length 2p whose center is in f_x
35             right-rotate(s, p)
```