Babenko, M., Gawrychowski, P., Kociumaka, T., Kolesnichenko, I., & Starikovskaia, T. (2016). Computing minimal and maximal suffixes of a substring. Theoretical Computer Science, 638, 112-121. DOI: 10.1016/j.tcs.2015.08.023

Computing minimal and maximal suffixes of a substring[☆]

Maxim Babenko[a], Paweł Gawrychowski[b,1], Tomasz Kociumaka[b,2], Ignat Kolesnichenko[c], Tatiana Starikovskaya[d]

[a] *National Research University Higher School of Economics, Myasnitskaya ul. 20, Moscow, Russia, 101000*
[b] *Institute of Informatics, University of Warsaw, Stefana Banacha 2, 02-097 Warsaw, Poland*
[c] *Moscow Institute of Physics and Technology, Institutskiy per. 9, Dolgoprudnyy, Russia, 141701*
[d] *University of Bristol, Senate House, Tyndall Avenue, Bristol, BS8 1TH, United Kingdom*

Abstract

We consider the problems of computing the maximal and the minimal non-empty suffixes of substrings of a longer text of length n. For the minimal suffix problem we show that for every τ, $1 \le \tau \le \log n$, there exists a linear-space data structure with $\mathcal{O}(\tau)$ query time and $\mathcal{O}(n \log n/\tau)$ preprocessing time. As a sample application, we show that this data structure can be used to compute the Lyndon decomposition of any substring of the text in $\mathcal{O}(k\tau)$ time, where k is the number of distinct factors in the decomposition. For the maximal suffix problem, we give a linear-space structure with $\mathcal{O}(1)$ query time and $\mathcal{O}(n)$ preprocessing time. In other words, we simultaneously achieve both the optimal query time and the optimal construction time.

1. Introduction

Computing the lexicographically maximal and minimal suffixes of a string is both an interesting problem on its own and a crucial ingredient in solutions to many other problems. For example, the famous constant-space pattern matching algorithm of Crochemore and Perrin and its more recent variants are based on the so-called critical factorizations, which can be derived from the maximal suffixes [1, 2].

The first non-trivial solution of the maximal and minimal suffix problems is due to Weiner, who introduced the suffix tree [3]. The suffix tree of a string can be constructed in linear time and occupies linear space. Once constructed, it allows to retrieve the maximal

[☆] This article is based on a study first reported at 24th and 25th Symposiums on Combinatorial Pattern Matching.

Email addresses: `maxim.babenko@gmail.com` (Maxim Babenko), `gawry@mimuw.edu.pl` (Paweł Gawrychowski), `kociumaka@mimuw.edu.pl` (Tomasz Kociumaka), `ignat@yandex-team.ru` (Ignat Kolesnichenko), `tat.starikovskaya@gmail.com` (Tatiana Starikovskaya)

[1] Currently holding a post-doc position at Warsaw Center of Mathematics and Computer Science.

[2] Supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program (Ministry of Science and Higher Education, Republic of Poland, grant number DI2012 01794).

and the minimal suffixes in constant time. Later, this result was improved by Duval [4] who showed that the suffixes can be found in linear time and constant additional space.

We consider a natural generalization of these problems. We assume that the strings we are asked to compute the maximal or the minimal suffixes for are actually substrings of a text T and that they are specified by their endpoints in T. Then, one can preprocess T and subsequently use this information to significantly speed up the computation of the desired suffixes of a query string. This seems to be a very natural setting whenever one thinks of storing large collections of static text data.

Let n be the length of T. We first show that for every τ, $1 \leq \tau \leq \log n$, there exists a linear-space data structure solving the minimal suffix problem with $\mathcal{O}(\tau)$ query time and $\mathcal{O}(\frac{n \log n}{\tau})$ preprocessing time. Secondly, we describe a linear-space data structure for the maximal suffix problem with $\mathcal{O}(1)$ query time which can be constructed in linear time. As a particular application, we show how to compute the Lyndon decomposition [5] of a substring of T in $\mathcal{O}(k\tau)$ time, where k is the number of distinct factors in the decomposition.

The key idea of our solution is to select, for each position j of the text T, a set of $\mathcal{O}(\log n)$ *canonical substrings* — substrings of T that end at j such that the lengths of two consecutive canonical substrings differ by a factor of at most 2. Note that for substrings with a fixed end-position, the maximal suffix becomes larger as the length of a substring increases, while the minimal suffix becomes smaller. Thus, for a query $x = T[i..j]$ we know that either the answer is the same as for the longest canonical suffix of x, or the resulting suffix is longer than $|x|/2$. For the latter case we develop a subroutine which exploits periodicities to compute the maximal (resp. minimal) suffix given its approximate length (within a factor of 2). The answers for canonical substrings are stored in $\mathcal{O}(\log n)$ bits for each position j. These bits let us to retrieve approximate lengths only; the exact answers are computed as in the previous case.

The basic $\mathcal{O}(n \log n)$-time construction algorithm computes the answers for all canonical substrings. However, for maximal suffixes we develop a linear-time construction algorithm. This is possible mainly due to the following fact: the length of the maximal suffix of a string cannot increase by more than one when a single letter is appended at the end. Minimal suffixes do not enjoy such a property, e.g., when `aa` is extended to `aab` the length of the minimal suffix increases from 1 to 3.

Related work. Text indexes that support various substring queries have been extensively studied in the literature. The study dates back to the invention of the suffix tree. Augmented properly, the suffix tree can be used to answer the substrings equality and the longest common prefix queries in constant time and linear space [6].

Cormode and Muthukrishnan [7] initiated a study on substring compression problems, where the goal is to quickly find the compressed representation or the compressed size for a given substring of the text. Some of their results were later improved in [8] and [9].

Recently, substring queries gained more attention. It has been shown that various periodicity-related queries can be answered in logarithmic or constant time [10, 11, 9]. Some of these results apply a linear-space data structure for internal pattern matching queries, which are to find all occurrences of one substring of the text in another substring [9]. Yet another type of substring queries is range LCP queries studied in [12, 13].

Queries asking for the k-th lexicographically smallest suffix of a substring, more general than both the minimal and the maximal suffix queries, have also been studied. They can be answered in $\mathcal{O}(\log n)$-time by a wavelet suffix tree, a linear space data structure which admits an $\mathcal{O}(n\sqrt{\log n})$-time construction algorithm [14]. However, wavelet suffix trees are less efficient and much more involved than the data structures we specifically design for minimal and maximal suffix queries.

2. Preliminaries

We start by introducing some standard notation and definitions. Let Σ be a finite non-empty set (called an *alphabet*). The elements of Σ are *letters*. A finite ordered sequence of letters (possibly empty) is called a *string*. Letters in a string are numbered starting from 1, that is, a string T of *length* k consists of letters $T[1], T[2], \ldots, T[k]$. The length of T is denoted by $|T|$. For $i \leq j$, $T[i..j]$ denotes the *substring* of T from position i to position j (inclusive). If $i = 1$ or $j = |T|$, then we omit these indices and we write $T[..j]$ and $T[i..]$. Substring $T[..j]$ is called a *prefix* of T, and $T[i..]$ is called a *suffix* of T.

A *border* of a string T is a string that is both a prefix and a suffix of T but differs from T. A string T is called *periodic with period* ρ if $T = \rho^s \rho'$ for an integer $s \geq 1$ and a (possibly empty) proper prefix ρ' of ρ. Borders and periods are dual notions: if T has period ρ, then it has a border of length $|T| - |\rho|$, and vice versa; see, e.g., [15].

Fact 1 ([16]). *If a string T has periods ρ and γ such that $|\rho| + |\gamma| \leq |T|$, then T has a period of length* $\gcd(|\rho|, |\gamma|)$*, the greatest common divisor of ρ and γ.*

Lemma 2. *If a string T has a proper border, then its shortest border has length at most $|T|/2$.*

Proof. Suppose that the shortest non-empty border of T has length larger than $|T|/2$, then by border-period duality T has a period ρ smaller than $|T|/2$. Since 2ρ is also a period and $2\rho < |T|$, we get another (shorter) border of T, a contradiction. □

We assume the word RAM model of computation [17] with word size $\Omega(\log n)$. Letters are treated as integers in range $\{1, \ldots, |\Sigma|\}$, so a pair of letters can be compared in $O(1)$ time. We also assume $\Sigma = n^{\mathcal{O}(1)}$ so that all letters of the input text T can be sorted in $\mathcal{O}(n)$ time. The natural linear order on Σ is extended in a standard way to the *lexicographic* order of strings over Σ. Namely, $T_1 \prec T_2$ if either

(a) T_1 is a prefix of T_2, or

(b) there exists $i < \min(|T_1|, |T_2|)$ such that $T_1[..i] = T_2[..i]$, and $T_1[i+1] < T_2[i+1]$.

Consider a fixed string T. For $i < j$ let $\mathit{Suf}[i, j]$ denote $\{T[i..], \dots, T[j..]\}$. The *suffix array* of a string T is a permutation SA on $\{1, \dots, |T|\}$ defining the lexicographic order on $\mathit{Suf}[1, |T|]$. More precisely, $SA[r] = i$ iff the rank of $T[i..]$ in the lexicographic order on $\mathit{Suf}[1, |T|]$ is r. For a string T, both SA and its inverse occupy linear space and can be constructed in linear time; see [18] for a survey.

When speaking of substrings $T[i..j]$ of a given fixed text T we assume, as long as this leads to no confusion, that the former are represented by the indices i and j.

Fact 3 ([15, 6, 19]). *Suffix array can be enhanced in linear time to answer the following queries in $\mathcal{O}(1)$ time:*

(a) Given substrings x, y of T, compute their longest common prefix $\mathrm{lcp}(x, y)$.

(b) Given substrings x, y of T, check if $x \prec y$.

(c) Given indices i, j, compute the maximal and the minimal suffixes in $\mathit{Suf}[i, j]$.

In particular, Fact 3(a) implies that given substrings x, y of T, it is possible to check in $\mathcal{O}(1)$ time if x is a prefix of y.

Lemma 4. *The following queries can also be answered in $\mathcal{O}(1)$ time using the enhanced suffix array: given substrings x, y of T, compute the largest integer α such that x^α is a prefix of y.*

Proof. It suffices to note that if x is a prefix of $y = T[i..j]$ (which can be determined in $\mathcal{O}(1)$ time), then $(\alpha - 1)|x| \le \mathrm{lcp}(T[i..j], T[i + |x|..j]) < \alpha|x|$. $\square$

Queries involving the enhanced suffix array of T^R, the reverse of T, are also meaningful in terms of T. In particular for a pair of substrings x, y of T we can compute their longest common suffix $\mathrm{lcs}(x, y)$ and the largest integer α such that x^α is a suffix of y.

3. Minimal Suffix

Consider a string T of length n. For each position j we select $\mathcal{O}(\log n)$ substrings $T[k..j]$, which we call *canonical*. By C_j^ℓ we denote the ℓ-th shortest canonical substring ending at position j. For a pair of integers $1 \le i < j \le n$, we define $\alpha(i, j)$ to be the largest integer ℓ such that C_j^ℓ is a proper suffix of $T[i..j]$. The following properties of canonical substrings are assumed:

(a) $C_j^1 = T[j..j]$ and for some $\ell = \mathcal{O}(\log n)$ we have $C_j^\ell = T[1..j]$,

(b) $\left|C_j^{\ell+1}\right| \le 2\left|C_j^\ell\right|$ for any ℓ,

(c) $\alpha(i, j)$ and $|C_j^\ell|$ are computable in $\mathcal{O}(1)$ time given i, j and ℓ, j respectively.

Our data structure works for any choice of canonical substrings satisfying these properties, including the simplest when $|C_j^\ell| = \min(2^{\ell-1}, j)$. The algorithm is based on two observations:

Lemma 5. *The minimal suffix of $T[i..j]$ is either equal to*

(a) $T[p..j]$, where p is the starting position of the minimal suffix in $Suf[i, j]$; or

(b) the shortest non-empty border of $T[p..j]$.

Proof. We shall prove that the minimal suffix $T[\mu..j]$ of $T[i..j]$ is both a prefix and a suffix of $T[p..j]$. Since $T[\mu..j]$ is the minimal suffix, it is smaller or equal to $T[p..j]$. By definition of the lexicographic order, either $T[\mu..j]$ is a prefix of $T[p..j]$, or there exists ℓ such that $T[\mu..\mu+\ell] = T[p..p+\ell]$ and $T[\mu+\ell+1] < T[p+\ell+1]$. If $T[\mu..j]$ is a prefix of $T[p..j]$, then we have $|T[\mu..j]| \leq |T[p..j]|$ and thus $T[\mu..j]$ is also a suffix of $T[p..j]$. Let us now show that the second case is impossible. Indeed, it follows that $T[\mu..] \prec T[p..]$, a contradiction.

We now know that $T[\mu..j] = T[p..j]$ or $T[\mu..j]$ is a non-empty border of $T[p..j]$. All borders of $T[p..j]$ are suffixes of $T[i..j]$, so in the latter case $T[\mu..j]$ must be a minimal non-empty border of $T[p..j]$. Because the lexicographic order on borders coincides with the order by lengths, this is also the shortest of these borders. □

Example 6. Consider a text $T =$ `cabacabaa` and its substrings $T[5..8]$ and $T[1..4]$, both equal to `caba`. For $T[5..8]$ we have $p = 7$ and $T[7..8] =$ `a` is the minimal suffix. On the other hand, for $T[1..4]$ we have $p = 2$ and the minimal suffix is the shortest border of $T[2..4] =$ `aba`.

Lemma 7. *The minimal suffix of $T[i..j]$ is either equal to*

(a) $T[p..j]$, where p is the starting position of the minimal suffix in $Suf[i, j]$; or

(b) the minimal suffix of $C_j^{\alpha(i,j)}$.

Proof. By Lemma 5, the minimal suffix is either equal to $T[p..j]$ or to its shortest non-empty border. In the latter case by Lemma 2 the length of the minimal suffix is at most $\frac{1}{2}|T[p..j]| \leq \frac{1}{2}|T[i..j]|$. Also property (b) of canonical substrings implies that $|C_j^{\alpha(i,j)}| \geq \frac{1}{2}|T[i..j]|$. Thus, in this case the minimal suffix of $T[i..j]$ is the minimal suffix of $C_j^{\alpha(i,j)}$. □

3.1. Data structure

The data structure, apart from the enhanced suffix array, contains, for each $j = 1, \ldots, n$, a bit vector B_j of length $\alpha(1, j)$. We set $B_j[\ell] = 1$ if and only if the minimal suffix of C_j^ℓ is longer than $C_j^{\ell-1}$ (equivalently, if C_j^ℓ and $C_j^{\ell-1}$ do not share a common minimal suffix). For $\ell = 1$ we always set $B_j[1] = 1$, as C_j^1 is the minimal suffix of itself. Recall that the number of canonical substrings for each j is $\mathcal{O}(\log n)$, so each B_j fits into a constant number of machine words, and thus the data structure takes $\mathcal{O}(n)$ space.

3.2. Queries

Assume we are looking for the minimal suffix of $T[i..j]$. First, compute $\alpha(i,j)$, which can be done in constant time. Next, find the minimal suffix $T[p..]$ in $Suf[i,j]$; using the enhanced suffix array this also takes constant time; see Fact 3. This gives us the first candidate $T[p..j]$.

Then, we examine the bit vector B_j to compute the minimal suffix of $C_j^{\alpha(i,j)}$. Let $\ell \le \alpha(i,j)$ be the largest index such that $B_j[\ell] = 1$. Note that such an index always exists (as $B_j[1] = 1$) and it can be found in constant time due to the following result.

Fact 8. *Given a bit vector B of $b = \mathcal{O}(\log n)$ bits and an index $k \le b$, the most significant bit position not exceeding k', i.e., value $\max\{k' \le k : B[k'] = 1\}$, can be found in $\mathcal{O}(1)$ time.*

Proof. Note that all standard arithmetic and bitwise operations can be performed in constant time on arguments of $\mathcal{O}(b) = \mathcal{O}(\log n)$ bits. In particular, we can perform bitwise `and` of B and $2^k - 1 = (1 \ll k) - 1$ to mask out bits at indices greater than k. The query now reduces to determining the most significant bit position in the resulting bit vector. As shown by Fredman and Willard [20], this can be achieved in constant time. Moreover, most modern processors provide such operation in the instruction set; see also [21]. $\square$

By definition of B_j, the minimal suffix of $C_j^{\alpha(i,j)}$ coincides with the minimal suffix of C_j^{ℓ}. Also, since $B_j[\ell] = 1$, case (a) of Lemma 7 holds for C_j^{ℓ}. This yields the second candidate $T[p'..j]$, where $T[p'..]$ is the minimal suffix in $Suf[j - |C_j^{\ell}| + 1, j]$.

Finally, we compare $T[p..j]$ with $T[p'..j]$ in constant time (relying on Fact 3) and output the smaller of these substrings. This completes the description of our constant-time query algorithm.

3.3. Construction

A simple $\mathcal{O}(n \log n)$-time construction algorithm also relies on Lemma 7. It suffices to show that, once the enhanced suffix array is built, we can determine B_j in $\mathcal{O}(\log n)$ time. We find the minimal suffix of C_j^{ℓ} for consecutive values of ℓ. Once we know the answer for $\ell - 1$, case (a) of Lemma 7 gives us the second candidate for the minimal suffix of C_j^{ℓ}, and the enhanced suffix array lets us choose the smaller of these two candidates. We set $B_j[\ell] = 1$ if the smaller candidate is longer than $C_j^{\ell-1}$. Therefore we obtain the following result.

Theorem 9. *A string T of length n can be stored in an $\mathcal{O}(n)$-space structure that computes the minimal suffix of a given substring of T in $\mathcal{O}(1)$ time. This data structure can be constructed in $\mathcal{O}(n \log n)$ time.*

The simple construction described above works for any choice of canonical substrings. However, to derive a trade-off between query and construction times, we consider a specific choice of canonical substrings and give an alternative construction method. Before we actually obtain the trade-off, let us describe the alternative construction algorithm in a basic $\mathcal{O}(n \log n)$-time variant.

It will be convenient to have many canonical substrings C_j^ℓ which are prefixes of each other, because then we can make use of Duval's algorithm (Algorithm 3.1 [4]) that computes the minimal suffixes of all prefixes of a string in linear time.

For $\ell = 1$ we define $C_j^1 = T[j..j]$. For $\ell > 1$ we set $m = \lfloor \ell/2 \rfloor - 1$ and define C_j^ℓ by

$$\left|C_j^\ell\right| = \begin{cases} 2 \cdot 2^m + (j \bmod 2^m) & \text{if } \ell \text{ is even,} \\ 3 \cdot 2^m + (j \bmod 2^m) & \text{otherwise.} \end{cases}$$

Note that if $2 \cdot 2^m \leq j < 3 \cdot 2^m$, then $T[1..j] = C_j^{2m+2}$, while if $3 \cdot 2^m \leq j < 4 \cdot 2^m$, then $T[1..j] = C_j^{2m+3}$; see Fig. 1. Therefore the number of canonical substrings ending at j is $\mathcal{O}(\log n)$.

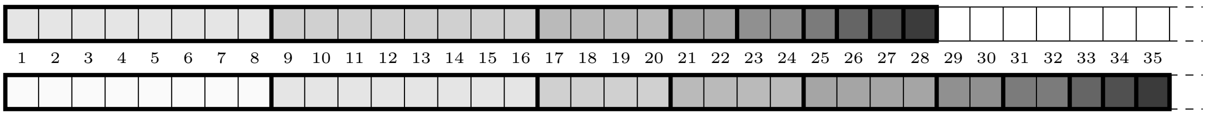

Figure 1: There are 9 canonical suffixes of $T[1..28]$: their lengths are $1, 2, 3, 4, 6, 8, 12, 20, 28$. On the other hand, $T[1..35]$ has 10 canonical substrings of lengths $1, 2, 3, 5, 7, 11, 15, 19, 27, 35$.

The following facts show that the above choice of canonical substrings satisfies properties (b) and (c).

Lemma 10 (Property (b)). *For any position j and value $\ell < \alpha(1, j)$, we have $\left|C_j^{\ell+1}\right| < 2\left|C_j^\ell\right|$.*

Proof. For $\ell = 1$ the statement holds trivially. Consider $\ell \geq 2$. Let m, as before, denote $\lfloor \ell/2 \rfloor - 1$. If ℓ is even, then $\ell + 1$ is odd and we have

$$\left|C_j^{\ell+1}\right| = 3 \cdot 2^m + (j \bmod 2^m) < 4 \cdot 2^m \leq 2 \cdot (2 \cdot 2^m + (j \bmod 2^m)) = 2\left|C_j^\ell\right|$$

while for odd ℓ

$$\left|C_j^{\ell+1}\right| = 2 \cdot 2^{m+1} + (j \bmod 2^{m+1}) < 3 \cdot 2^{m+1} \leq 2 \cdot (3 \cdot 2^m + (j \bmod 2^m)) = 2\left|C_j^\ell\right|.$$

□

Lemma 11 (Property (c)). *For $1 \leq i < j \leq n$, value $\alpha(i, j)$ can be computed in constant time.*

Proof. Let $m = \lfloor \log |T[i..j]| \rfloor$. Observe that

$$\begin{aligned} \left|C_j^{2m-1}\right| &= 3 \cdot 2^{m-2} + (j \bmod 2^{m-2}) < 2^m \leq |T[i..j]| \\ \left|C_j^{2m+2}\right| &= 2 \cdot 2^m + (j \bmod 2^m) \geq 2^{m+1} > |T[i..j]|. \end{aligned}$$

Thus $\alpha(i, j) \in \{2m-1, 2m, 2m+1\}$, and we pick the correct value in constant time. □

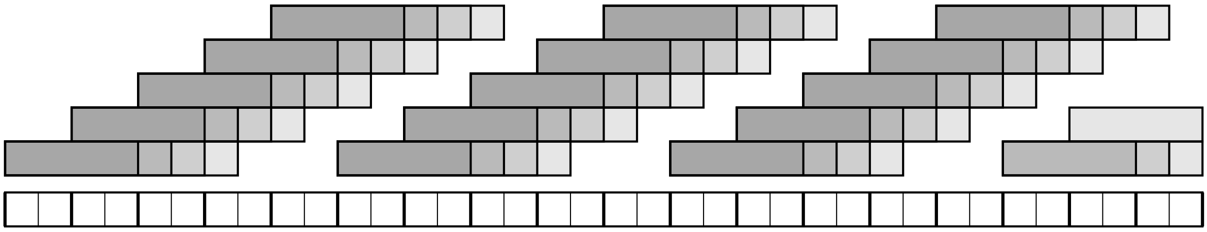

Figure 2: Canonical substring C_j^4 and C_j^5 (corresponding to $m = 1$) start at odd positions and have lengths between 4 and 7. To compute their minimal suffixes, we run Duval's algorithms for each four consecutive chunks of length two (and for the last three and the last two chunks).

After building the enhanced suffix array, we set all bits $B_j[1]$ to 1. Then for each $\ell > 1$ we compute the minimal suffixes of the substrings C_j^ℓ as follows. Fix $\ell > 1$ and split T into *chunks* of size 2^m each, where $m = \lfloor \ell/2 \rfloor - 1$. Now each C_j^ℓ is a prefix of a concatenation of at most four such chunks. We run Duval's algorithm for each four (or less at the end) consecutive chunks. This gives the minimal suffixes of C_j^ℓ for all positions j in $\mathcal{O}(n)$ time; see Fig. 2. The value $B_j[\ell]$ is determined by comparing the length of the computed minimal suffix of C_j^ℓ with $|C_j^{\ell-1}|$. We have $\mathcal{O}(\log n)$ phases, which gives $\mathcal{O}(n \log n)$ total time complexity and $\mathcal{O}(n)$ total space consumption.

3.4. Trade-off

To obtain a data structure with $\mathcal{O}(n \log n/\tau)$-time construction and $\mathcal{O}(\tau)$-time queries, we define the bit vectors in a slightly different way. We set B_j^τ to be of size $\lfloor \alpha(1,j)/\tau \rfloor$ with $B_j^\tau[k] = 1$ if and only if $k = 1$ or the minimal suffix of $C_j^{\tau k}$ is longer than $C_j^{\tau(k-1)}$. This way we need only $\mathcal{O}(\log n/\tau)$ phases in the construction algorithm, so it takes $\mathcal{O}(n \log n/\tau)$ time.

Again, assume we are looking for the minimal suffix of $T[i..j]$. As before, the difficult part is to find the minimal suffix of $C_j^{\alpha(i,j)}$. Our goal is to compute $\ell \le \alpha(i,j)$ such that the minimal suffix of $C_j^{\alpha(i,j)}$ coincides with the minimal suffix of C_j^ℓ, but is longer than $C_j^{\ell-1}$.

If we knew that $\alpha(i,j) = \tau k$ for an integer k, we could find the largest $k' \le k$ such that $B^\tau[k'] = 1$ and we would know that $\ell \in (\tau(k'-1), \tau k']$. In general, we choose the largest k such that $\tau k \le \alpha(i,j)$, and then we know that we consider all $\ell \in (\tau k, \alpha(i,j)] \cap (\tau(k'-1), \tau k']$, with k' defined as in the previous special case.

In total we have $\mathcal{O}(\tau)$ possible values of ℓ, and we are guaranteed that the suffix we seek can be obtained using case (a) of Lemma 7 for C_j^ℓ for one of these values. After generating all these candidates we use the enhanced suffix array to find the smallest suffix among them. In total, queries take $\mathcal{O}(\tau)$ time thus proving the following result:

Theorem 12. *For every τ, $1 \le \tau \le \log n$, a string T of length n can be stored in an $\mathcal{O}(n)$-space data structure that computes in $\mathcal{O}(\tau)$ time the minimal suffix of a given substring of T. This data structure can be constructed in $\mathcal{O}(n \log n/\tau)$ time.*

3.5. Lyndon decompositions

As a corollary we obtain an efficient data structure for computing Lyndon decompositions of substrings of T. Recall that a string w is a *Lyndon word* if it is strictly smaller than its proper cyclic rotations. For a non-empty string x, a decomposition $x = w_1^{\alpha_1} w_2^{\alpha_2} \dots w_k^{\alpha_k}$ is called a *Lyndon decomposition* if and only if $w_1 > w_2 > \cdots > w_k$ are Lyndon words [5]. Every string admits a unique Lyndon decomposition, which can be obtained as follows; see [4]. The last factor w_k is the minimal suffix of x and $w_k^{\alpha_k}$ is the largest power of w_k which is a suffix of x. Also, $w_1^{\alpha_1} w_2^{\alpha_2} \dots w_{k-1}^{\alpha_{k-1}}$ is the Lyndon decomposition of the remaining prefix of x. The last factor $w_k^{\alpha_k}$ can be computed in constant time using Theorem 12 and Lemma 4, which yields the following corollary.

Corollary 13. *For every τ, $1 \le \tau \le \log n$, a string T of length n can be stored in an $\mathcal{O}(n)$-space data structure that computes the Lyndon decomposition of a given substring of T in $\mathcal{O}(k\tau)$ time, where k is the number of distinct factors in the decomposition. This data structure can be constructed in $\mathcal{O}(n \log n/\tau)$ time.*

4. Maximal Suffix

Our data structure for the maximal suffix problem is very similar to the one we have developed for the minimal suffix. In particular, it is defined for canonical substrings C_j^{ℓ} satisfying the same three properties. However, in contrast to the minimal suffix problem, the properties specific to maximal suffixes let us design a linear-time construction algorithm.

The only component of Section 3 which cannot be immediately adapted to the maximal suffix problem is Lemma 7. While its exact counterpart is not true, in Section 4.1 we prove the following statement, which is equivalent in terms of algorithmic applications. The proof is rather involved, but it yields an relatively simple algorithm, which asks a few queries to the enhanced suffix array (provided by Fact 3 and Lemma 4).

Lemma 14. *Consider a substring $T[i..j]$. Using the enhanced suffix array of T, one can compute in $\mathcal{O}(1)$ time an index p ($i \le p \le j$) such that the maximal suffix of $T[i..j]$ is either equal to*

(a) $T[p..j]$; or

(b) the maximal suffix of $C_j^{\alpha(i,j)}$.

Just as in the data structure described in Section 3, apart from the enhanced suffix array, we store bit vectors B_j, $j \in [1, n]$, with $B_j[\ell] = 1$ if $\ell = 1$ or the maximal suffix of C_j^{ℓ} is longer than $C_j^{\ell-1}$. The query algorithm described in Section 3.2 can be adapted in an obvious way, i.e., so that it uses Lemma 14 instead of Lemma 7 and chooses the larger of the two candidates as the answer. This shows the following theorem:

Theorem 15. *A string T of length n can be stored in an $\mathcal{O}(n)$-space structure that enables to compute the maximal suffix of any substring of T in $\mathcal{O}(1)$ time.*

The $\mathcal{O}(n \log n)$-time construction algorithms and the trade-off between query and construction time, described in Sections 3.3 and 3.4, are also easy to adapt to the maximal suffix problem. They are, however, outperformed by a $\mathcal{O}(n)$-time construction presented in Section 4.2.

4.1. Proof of Lemma 14

Below we describe a constant-time algorithm, which returns a position $p \in [i, j]$. If the maximal suffix $T[\mu..j]$ of $T[i..j]$ is shorter than $C_j^{\alpha(i,j)}$ (case (b) of Lemma 14), the algorithm may return any $p \in [i, j]$. Hence, we assume that $T[\mu..j]$ is longer than $C_j^{\alpha(i,j)}$ and show that under this assumption the algorithm returns $p = \mu$. Suppose $T[p_1..]$ is the maximal suffix within $Suf[i, j - |C_j^{\alpha(i,j)}|]$.

Observation 16. *$P_1 = T[p_1..j]$ is a prefix of $T[\mu..j]$.*

Proof. The proof is by contradiction. Suppose that the first ℓ letters of the suffixes are equal, but $T[p_1 + \ell] \neq T[\mu + \ell]$. From the definition of the lexicographic order and $T[p_1..j] \preceq T[\mu..j]$ we obtain $T[p_1 + \ell] < T[\mu + \ell]$. But then $T[p_1..] \prec T[\mu..]$, i.e. $T[\mu..]$ is another suffix in $Suf[i, j - |C_j^{\alpha(i,j)}|]$ which is larger than $T[p_1..]$, a contradiction. □

If $p_1 = i$, then we must have $\mu = i$ as well. Otherwise, we define p_2 so that $T[p_2..]$ is maximal within $Suf[i, p_1 - 1]$.

Lemma 17. *If P_1 is not a prefix of $P_2 = T[p_2..j]$, then $\mu = p_1$. Otherwise P_2 is a prefix of $T[\mu..j]$.*

Proof. We consider two cases depending on whether $Suf[i, p_1 - 1]$ contains a suffix that starts with P_1 or not. If no suffix in $Suf[i, p_1 - 1]$ starts with P_1, then $\mu \notin [i, p_1 - 1]$ (as $T[\mu..j]$ starts with P_1). Consequently, $\mu \geq p_1$. But $T[p_1..j]$ is a prefix of $T[\mu..j]$, i.e. the latter cannot be shorter than $T[p_1..j]$ and therefore $\mu = p_1$.

Now consider the second case. Let Q be the prefix of P_2 of length $|P_1|$. If $Q \succ P_1$, then $P_2 = QT[p_2 + |P_1|..j] \succ P_1 T[\mu + |P_1|..j] = T[\mu..j]$, which is a contradiction. If $Q \prec P_1$, then no suffix in $Suf[i, p_1 - 1]$ can start with P_1 for otherwise such a suffix would be larger than P_2. Therefore, $Q = P_1$.

If $P_2 = T[\mu..j]$, then $p_2 = \mu$ and the lemma follows. Otherwise $P_2 \prec T[\mu..j]$. Suppose that P_2 is not a prefix of $T[\mu..j]$. Then $T[p_2..p_2+\ell] = T[\mu..\mu+\ell]$ and $T[p_2+\ell+1] < T[\mu+\ell+1]$ for some $\ell \geq |P_1|$. Therefore $T[p_2..] \prec T[\mu..]$ and $T[p_2..]$ is not the maximal suffix in $Suf[i, p_1 - 1]$, a contradiction. □

Lemma 18. *The shortest period of P_2 is $\rho = T[p_2..p_1 - 1]$.*

Proof. Clearly, P_1 is a border of P_2. Consequently, $\rho = T[p_2..p_1 - 1]$ is a period of P_2. It remains to prove that ρ is the shortest period. Suppose the opposite holds and let γ be the shortest period of P_2. The properties of canonical substrings imply that lengths of any two suffixes of $T[i..j]$ starting in $[i, j - |C_j^{\alpha(i,j)}|]$ differ by at most a factor of two. In particular, $|P_2| \leq 2|P_1|$. Therefore $|\gamma| + |\rho| < 2|\rho| \leq |T[p_2..j]|$ and by Periodicity Lemma (Fact 1) P_2 has a period of length $\gcd(|\gamma|, |\rho|)$. Since γ is the shortest period, $|\rho|$ must be a multiple of $|\gamma|$, i.e., $\rho = \gamma^k$ for some $k \geq 2$.

Consider the string $\gamma T[p_1..]$ and compare it with $T[p_1..]$. Clearly, these two strings are not equal. We now show that neither $T[p_1..] \prec \gamma T[p_1..]$ or $T[p_1..] \succ \gamma T[p_1..]$ is possible.

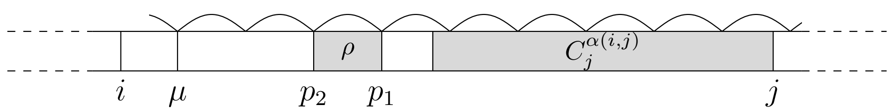

Figure 3: A schematic illustration of Lemma 19.

First suppose that $T[p_1..] \prec \gamma T[p_1..]$. Prepending both parts of the latter inequality by copies of γ gives $\gamma^{\ell-1}T[p_1..] \prec \gamma^{\ell}T[p_1..]$ for any $1 \le \ell \le k$. From transitivity of $\prec$ it follows that $T[p_1..] \prec \gamma^k T[p_1..] = T[p_2..]$, which contradicts the maximality of $T[p_1..]$ in $\mathit{Suf}[i, j - |C_j^{\alpha(i,j)}|]$.

Now suppose that $T[p_1..] \succ \gamma T[p_1..]$, which implies $\gamma^{k-1}T[p_1..] \succ \gamma^k T[p_1..]$. But $\gamma^{k-1}T[p_1..] = T[p_2 + |\gamma|..]$ and $\gamma^k T[p_1..] = T[p_2..]$, so $T[p_2 + |\gamma|..]$ is larger than $T[p_2..]$ and belongs to $\mathit{Suf}[i, p_1 - 1]$, a contradiction. $\square$

Lemma 19. *$T[\mu..j]$ is the longest suffix of $T[i..j]$ equal to $\rho^r\rho'$ for some integer r; see Fig. 3.*

Proof. Clearly, P_2 is a border of $T[\mu..j]$. Again, from the properties of canonical substrings we have $|T[\mu..j]| \le 2|P_1|$. Therefore, $|T[\mu..j]| + |\rho| \le 2|P_1| + |\rho| \le 2|P_2|$. This inequality implies that the occurrences of P_2 as a prefix and as a suffix of $T[\mu..j]$ have an overlap of at least $|\rho|$ positions. Since $|\rho|$ is a period of P_2, $|\rho|$ is also a period of $T[\mu..j]$.

Thus $T[\mu..j] = \rho''\rho^r\rho'$, where r is an integer and ρ'' is a proper suffix of ρ. Furthermore, ρ^2 is a prefix of $T[\mu..j]$, since it is a prefix of P_2, which is in turn a prefix of $T[\mu..j]$. If ρ'' is not an empty string, there is a non-trivial occurrence of ρ in ρ^2, which contradicts ρ being the shortest period of P_2; see, e.g., [15]. The claim follows. Note also that r must be the maximal possible, since for every $t > r$ we have $\rho^t\rho' \succ \rho^r\rho'$. $\square$

Proof (of Lemma 14). Let $T[p_1..]$ be the maximal suffix in $\mathit{Suf}[i, j-|C_j^{\alpha(i,j)}|]$ and $T[p_2..]$ be the maximal suffix in $\mathit{Suf}[i, p_1-1]$. We first compute p_1. If $p_1 = i$, we return i. Otherwise, we compute p_2 and check whether $T[p_1..j]$ is a prefix of $T[p_2..j]$. If not, we return $p = p_1$. Otherwise, we determine the largest integer r such that ρ^r, where $\rho = T[p_2..p_1-1]$, is a suffix of $T[i..p_1-1]$, and return $p = p_1 - r|\rho|$; see Fig. 4. Each of the steps takes constant time by Fact 3 and Lemma 4. Correctness of the algorithm follows from the discussion above. $\square$

4.2. Construction

For $1 \le p \le j \le n$ we say that a position p is *j-active* if there is no position $p' \in [p+1, j]$ such that $T[p..j] \prec T[p'..j]$. In these terms, the starting position of the maximal suffix of $T[i..j]$ is the leftmost j-active position in $[i, j]$. The definition also implies that for any $\ell > 1$ we have $B_j[\ell] = 1$ if and only if there is at least one j-active position within the range $R_j^\ell = \left[j - |C_j^\ell| + 1, j - |C_j^{\ell-1}|\right]$. We set $R_j^1 = [j, j]$ so that this equivalence also holds for $\ell = 1$ (since j is always j-active).

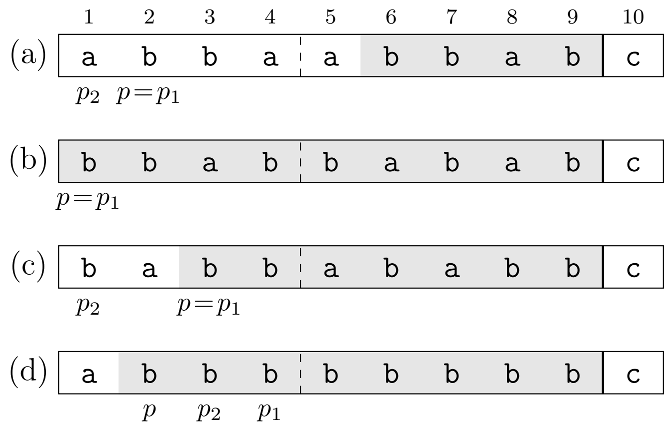

Figure 4: An illustration of cases which may occur in Lemma 14. In all four examples we are looking for the maximal suffix of $x = T[1..9]$ and its (unknown) maximal suffix is shaded. We assume that $C_9^{\alpha(1,9)} = T[5..9]$. (a) We have $p = 2$ but the maximal suffix of $C_9^{\alpha(1,9)}$ is larger than $T[2..9]$. (b) We have $p_1 = i = 1$, so $p = 1$. (c) We have $p_1 = 3$ and $p_2 = 1$. However, $T[3..9]$ is not a prefix of $T[1..9]$, so $p = p_1 = 3$. (d) We have $p_1 = 4$, $p_2 = 3$ and $T[3..9]$ is a prefix of $T[2..9]$. Hence, $\rho = \mathtt{b}$ is a period of $T[2..9]$. This period continues to the left until position $p = 2$.

Example 20. If $T[1..8] = \mathtt{dcccabab}$, the 8-active positions are $1, 2, 3, 4, 6, 8$. Consider, for example, $p = 3$. We have that $T[3..8] = cabab$ and this string is the maximal suffix of itself.

Our construction algorithm iterates over $j = 1..n$, maintaining the list of active positions and computing the bit vectors B_j. We also maintain the ranges R_j^ℓ for the choice of canonical substrings defined in Section 3.3, which form a partition of $[1, j]$. The following two results describe the changes of the list of j-active positions and the ranges R_j^ℓ when we increment j.

Lemma 21. *If the list of all $(j-1)$-active positions consists of $p_1 < p_2 < \cdots < p_z$, the list of j-active positions can be created by adding j, and repeating the following procedure: if p_k and p_{k+1} are two neighbours on the current list and $T[p_k..j] \prec T[p_{k+1}..j]$, remove p_k from the list. The latter may happen only if* $\mathrm{lcp}(T[p_k..], T[p_{k+1}..]) = j - p_{k+1}$.

Proof. First, note that if a position $1 \le p \le j-1$ is not $(j-1)$-active, then it is not j-active either. Indeed, if p is not $(j-1)$-active, then by the definition there is a position $p < p' \le j-1$ such that $T[p..j-1] \prec T[p'..j-1]$. Consequently, $T[p..j] = T[p..j-1]T[j] \prec T[p'..j-1]T[j] = T[p'..j]$ and p is not j-active. Hence, the only candidates for j-active positions are the $(j-1)$-active positions and j.

All elements removed by our procedure clearly fail to be j-active. Thus, when it terminates, the contents of the list form a superset of the set of j-active positions. Moreover, the suffixes $T[p..j]$ starting at positions p on the list form a lexicographically decreasing sequence. We shall prove that each of these positions is j-active. For a proof by contradiction suppose this is not the case and some index p in the list is not j-active. Let $T[p'..j]$ be maximal suffix of $T[p..j]$. Then $p' > p$ is j-active and satisfies $T[p..j] \prec T[p'..j]$. This contradicts the monotonicity combined with the fact that the list contains all j-active positions.

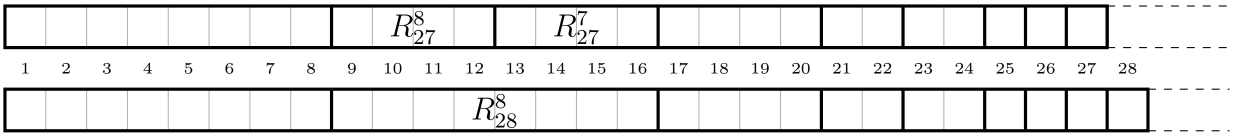

Figure 5: The partitions of $[1, j]$ into R_j^ℓ for $j = 27$ and $j = 28$. As for $j = 28$ we have $k = 2$ and $2k+4 = 8$, R_{27}^7 and R_{27}^8 are merged into R_{28}^8.

Finally, let us justify the last part of the statement. Note that p_k is $(j-1)$-active but not j-active. Thus, $T[p_k..j] \prec T[p_{k+1}..j]$ but $T[p_k..j-1] \succ T[p_{k+1}..j-1]$. This may only be true when $T[p_{k+1}..j-1]$ is a prefix (and therefore a border) of $T[p_k..j-1]$ or when $p_{k+1} = j$ and $T[p_{k+1}..j-1]$ is empty. In both cases we obtain $\mathrm{lcp}(T[p_k..j], T[p_{k+1}..j]) = j - p_{k+1}$ which is equivalent to $\mathrm{lcp}(T[p_k..], T[p_{k+1}..]) = j - p_{k+1}$. $\square$

Example 22. Let $T = \texttt{dcccababb}$. The 8-active positions are $1, 2, 3, 4, 6, 8$. The list of the 9-active positions is created by adding 9 and deleting 6. The latter is deleted because 6 and 8 are neighbours and $T[6..9] = \texttt{babb} \prec T[8..9] = \texttt{bb}$. Therefore, the 9-active positions are $1, 2, 3, 4, 8, 9$.

We now need a technical lemma which describes how the ranges R_j^ℓ are related to ranges R_{j-1}^ℓ; see Fig. 5 for an example.

Lemma 23. *Let $j \in [1, n]$ and assume 2^k is the largest power of two dividing j.*

(a) If $\ell = 1$, then $R_j^\ell = [j, j]$.

(b) If $2 \le \ell < 2k+4$, then $R_j^\ell = R_{j-1}^{\ell-1}$.

(c) If $\ell = 2k+4$, then $R_j^\ell = R_{j-1}^\ell \cup R_{j-1}^{\ell-1}$.

(d) If $\ell > 2k+4$, then $R_j^\ell = R_{j-1}^\ell$.

Proof. Observe that we have $R_j^1 = [j, j]$ and $R_j^2 = [j-1, j-1]$, while for $\ell > 2$

$$R_j^\ell = \begin{cases} [2^m(\lfloor \frac{j}{2^m} \rfloor - 2) + 1, 2^{m-1}(\lfloor \frac{j}{2^{m-1}} \rfloor - 3)] & \text{if } \ell \text{ is even,} \\ [2^m(\lfloor \frac{j}{2^m} \rfloor - 3) + 1, 2^m(\lfloor \frac{j}{2^m} \rfloor - 2)] & \text{otherwise,} \end{cases}$$

where $m = \lfloor \ell/2 \rfloor - 1$. Also note that

$$2^m(\lfloor \tfrac{j}{2^m} \rfloor - 3) = \begin{cases} 2^m(\lfloor \frac{j-1}{2^m} \rfloor - 2) & \text{if } 2^m \mid j \\ 2^m(\lfloor \frac{j-1}{2^m} \rfloor - 3) & \text{otherwise,} \end{cases}$$

$$2^m(\lfloor \tfrac{j}{2^m} \rfloor - 2) = \begin{cases} 2^{m-1}(\lfloor \frac{j-1}{2^{m-1}} \rfloor - 3) & \text{if } 2^m \mid j \\ 2^m(\lfloor \frac{j-1}{2^m} \rfloor - 2) & \text{otherwise.} \end{cases}$$

Moreover, $2^m \mid j \iff \ell \le 2k+3$ and $2^{m-1} \mid j \iff \ell \le 2k+5$, which makes it easy to check the claimed formulas. Note that it is possible that R_j^ℓ is defined only for values ℓ smaller than $2k+4$. This is exactly when the number of ranges grows by one, otherwise it remains unchanged. $\square$

We scan T from left to right and compute the bit vectors while maintaining the list of active positions and the partition of $[1, j]$ into ranges R_j^ℓ. Additionally, for every such range we have a counter storing the number of active positions inside. Recall that $B_j[\ell] = 1$ exactly when the ℓ-th counter is nonzero.

To efficiently update the list of active positions we store pointers to pairs of neighbouring positions. Whenever a new pair of neighbouring positions p_k, p_{k+1} appears, we insert a pointer to the pair into the list associated with a position $p_{k+1} + \mathrm{lcp}(T[p_k..], T[p_{k+1}..])$. (Remember that $\mathrm{lcp}(T[p_k..], T[p_{k+1}..])$ can be computed in constant time by Fact 3.)

Suppose that we already know the list of $(j-1)$-active positions, the bit vector B_{j-1}, and the number of $(j-1)$-active positions in each range R_{j-1}^ℓ. At the moment we reach j, we first update the list of $(j-1)$-active positions. We append j and then we process pointers stored in the list of neighbouring positions associated with position j. For a pointer to a pair (p_k, p_{k+1}) we check if p_k and p_{k+1} are still neighbours. If they are and $T[j+p_k-p_{k+1}] < T[j]$, we remove p_k from the list of active positions, otherwise we do nothing. If a position p is deleted from the list, we find the range it belongs to ($R_j^{\alpha(p-1,j)+1}$), and decrement the counter of active positions there. If a counter becomes zero, we clear the corresponding bit of the bit vector.

Next, we update the partition: first, we append a new range $[j, j]$ to the partition of $[1..j-1]$ and initialize its counter of active positions to one. Let 2^k be the largest power of two dividing j. We update the first $2k+4$ ranges using Lemma 23, including the counters and the bit vector. This takes $\mathcal{O}(k)$ time which amortizes to $\mathcal{O}(\sum_{k=1}^{\infty} \frac{k}{2^k}) = \mathcal{O}(1)$ over all values of j. Correctness of the algorithm follows from Lemmas 21 and 23.

Theorem 24. *A string T of length n can be stored in an $\mathcal{O}(n)$-space structure that in $\mathcal{O}(1)$ time computes the maximal suffix of a given substring of T. The data structure can be constructed in $\mathcal{O}(n)$ time.*

References

[1] M. Crochemore, D. Perrin, Two-way string-matching, J. ACM 38 (3) (1991) 650–674.

[2] D. Breslauer, R. Grossi, F. Mignosi, Simple real-time constant-space string matching, Theor. Comput. Sci. 483 (2013) 2–9.

[3] P. Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 1–11.

[4] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (4) (1983) 363–381.

[5] K. T. Chen, R. H. Fox, R. C. Lyndon, Free differential calculus, IV. The quotient groups of the lower central series, The Annals of Mathematics 68 (1) (1958) 81–95.

[6] M. A. Bender, M. Farach-Colton, The LCA problem revisited, in: G. H. Gonnet, D. Panario, A. Viola (Eds.), Latin American Symposium on Theoretical Informatics, LATIN 2000, Vol. 1776 of LNCS, Springer Berlin Heidelberg, 2000, pp. 88–94.

[7] G. Cormode, S. Muthukrishnan, Substring compression problems, in: 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, SIAM, 2005, pp. 321–330.

[8] O. Keller, T. Kopelowitz, S. L. Feibish, M. Lewenstein, Generalized substring compression, Theor. Comput. Sci. 525 (2014) 45–54.

[9] T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, Internal pattern matching queries in a text and applications, in: P. Indyk (Ed.), 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 532–551.

[10] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, Extracting powers and periods in a word from its runs structure, Theor. Comput. Sci. 521 (2014) 29–41.
[11] T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, Efficient data structures for the factor periodicity problem, in: L. Calderón-Benavides, C. N. González-Caro, E. Chávez, N. Ziviani (Eds.), String Processing and Information Retrieval, SPIRE 2012, Vol. 7608 of LNCS, Springer Berlin Heidelberg, 2012, pp. 284–294.
[12] A. Amir, A. Apostolico, G. M. Landau, A. Levy, M. Lewenstein, E. Porat, Range LCP, J. Comput. Syst. Sci. 80 (7) (2014) 1245–1253.
[13] M. Patil, R. Shah, S. V. Thankachan, Faster range LCP queries, in: O. Kurland, M. Lewenstein, E. Porat (Eds.), String Processing and Information Retrieval, SPIRE 2013, Vol. 8214 of LNCS, Springer International Publishing, 2013, pp. 263–270.
[14] M. Babenko, P. Gawrychowski, T. Kociumaka, T. Starikovskaya, Wavelet trees meet suffix trees, in: P. Indyk (Ed.), 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 572–591.
[15] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[16] N. J. Fine, H. S. Wilf, Uniqueness theorems for periodic functions, P. Am. Math. Soc. 16 (1) (1965) pp. 109–114.
[17] T. Hagerup, Sorting and searching on the word RAM, in: M. Morvan, C. Meinel, D. Krob (Eds.), Symposium on Theoretical Aspects of Computer Science, STACS 1998, Vol. 1373 of LNCS, Springer, Berlin Heidelberg, 1998, pp. 366–398.
[18] S. J. Puglisi, W. F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2).
[19] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: A. Amir, G. M. Landau (Eds.), Combinatorial Pattern Matching, CPM 2001, Vol. 2089 of LNCS, Springer Berlin Heidelberg, 2001, pp. 181–192.
[20] M. L. Fredman, D. E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436.
[21] M. Pătraşcu, M. Thorup, Dynamic integer sets with optimal rank, select, and predecessor search, in: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, IEEE Computer Society, 2014, pp. 166–175.