Computing minimal and maximal suffixes of a substring revisited

Maxim Babenko[1], Paweł Gawrychowski[2],
Tomasz Kociumaka[3], and Tatiana Starikovskaya[1⋆]

[1] National Research University Higher School of Economics (HSE)
[2] Max-Planck-Institut für Informatik
[3] Institute of Informatics, University of Warsaw

Abstract. We revisit the problems of computing the maximal and the minimal non-empty suffixes of a substring of a longer text of length n, introduced by Babenko, Kolesnichenko and Starikovskaya [CPM'13]. For the minimal suffix problem we show that for any $1 \leq \tau \leq \log n$ 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, i.e., we manage to achieve both the optimal query and the optimal construction time simultaneously.

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. As an example of the former, a well-known result by Duval [5] is that the maximal and the minimal suffixes of a string can be found in linear time and constant additional space. As an example of the latter, the famous constant space pattern matching algorithm of Crochemore-Perrin is based on the so-called critical factorizations, which can be derived from the maximal suffixes [4].

We consider a natural generalization of the problems. We assume that the string we are asked to compute the maximal or the minimal suffixes for is actually a substring of a text T of length n given in advance. 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 we are thinking about storing large collections of text data.

The problems of computing the minimal non-empty and the maximal suffixes of a substring of T were introduced in [1]. The authors proposed two linear-space data structures for T. Using the first data structure, one can compute the minimal suffix of any substring of T in $\mathcal{O}(\log^{1+\varepsilon} n)$ time. The second data structure

⋆ Tatiana Starikovskaya was partly supported by Dynasty Foundation.

allows to compute the maximal suffix of a substring of T in $\mathcal{O}(\log n)$ time. Here we improve upon both of these results. We first show that for any $1 \leq \tau \leq \log n$ there exists a linear-space data structure with $\mathcal{O}(\tau)$ query time and $\mathcal{O}(n \log n/\tau)$ preprocessing time solving the minimal suffix problem. Secondly, we describe a linear-space data structure for the maximal suffix problem with $\mathcal{O}(1)$ query time. The data structure can be constructed in linear time. Computing the minimal or the maximal suffix is a fundamental ingredient of complex algorithms, so our results can hopefully be used to efficiently solve other problems in such setting, i.e., when we are working with substrings of some long text T. As a particular application, we show how to compute the Lyndon decomposition [3] of a substring of T in $\mathcal{O}(k\tau)$ time, where k is the number of distinct factors in the decomposition.

2 Preliminaries

We start by introducing some standard notation and definitions. 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. When this leads to no confusion, the length of ρ will also be called a period of T. Borders and periods are dual notions; namely, if T has period ρ then it has a border of length $|T| - |\rho|$, and vice versa (see, e.g., [4]). A string T_1 is lexicographically smaller than T_2 ($T_1 \prec T_2$) if either (i) T_1 is a proper prefix of T_2; or (ii) there exists $0 \leq i < \min(|T_1|, |T_2|)$ such that $T_1[1..i] = T_2[1..i]$, and $T_1[i+1] < T_2[i+1]$.

Consider a fixed string T. For $i < j$ let $\mathit{Suf}[i, j]$ denote $\{T[i..], \ldots, T[j..]\}$. The set $\mathit{Suf}[1, |T|]$ of all non-empty suffixes of T is also denoted as Suf. The suffix array and the inverse suffix array of a string T are denoted by SA and ISA respectively. Both SA and ISA occupy linear space and can be constructed in linear time (see [6] for a survey). For strings x, y we denote the length of their longest common prefix by $\mathrm{lcp}(x, y)$. SA and ISA can be enhanced in linear time [4, 2] to answer the following queries in $\mathcal{O}(1)$ time:

(a) Given substrings x, y of T compute $\mathrm{lcp}(x, y)$ and determine if $x \prec y$.
(b) Given indices i, j compute the *maximal* and *minimal* suffix in $\mathit{Suf}[i, j]$.

Moreover, the enhanced suffix array can also be used to answer the following queries in constant time. Given substrings x, y of T compute the largest integer α such that x^α is a prefix of y. Indeed, it suffices to note that if x is a prefix of $y = T[i..j]$, then $(\alpha - 1)|x| \leq \mathrm{lcp}(T[i..j], T[i + |x|..j]) < \alpha|x|$. Queries on 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 S_j^ℓ we denote the ℓ-th shortest canonical

substring ending at the position j. For a pair of integers $1 \leq i < j \leq n$, we define $\alpha(i,j)$ to be the largest integer ℓ such that S_j^ℓ is a proper suffix of $T[i..j]$. We require the following properties of canonical substrings:

(a) $S_j^1 = T[j..j]$ and for some $\ell = \mathcal{O}(\log n)$ we have $S_j^\ell = T[1..j]$,
(b) $\left|S_j^{\ell+1}\right| \leq 2\left|S_j^\ell\right|$ for any ℓ,
(c) $\alpha(i,j)$ and $|S_j^\ell|$ 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 one with $|S_j^\ell| = \min(2^{\ell-1}, j)$. Our solution is based on the following lemma:

Lemma 1. *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 $Suf[i,j]$,
(b) or the minimal suffix of $S_j^{\alpha(i,j)}$.

Moreover, p can be found in $\mathcal{O}(1)$ time using the enhanced suffix array of T.

Proof. By Lemma 1 in [1] the minimal suffix is either equal to $T[p..j]$ or to its shortest non-empty border. Moreover, in the latter case the length of the minimal suffix is at most $\frac{1}{2}|T[p..j]| \leq \frac{1}{2}|T[i..j]|$. On the other hand, from the property (b) of canonical substrings we have that the length of $S_j^{\alpha(i,j)}$ is at least $\frac{1}{2}|T[i..j]|$. Thus, in the second case the minimal suffix of $T[i..j]$ is the minimal suffix of $S_j^{\alpha(i,j)}$. Note that for $i = j$ the values $\alpha(i,j)$ are not well-defined, but then case (a) holds. To prove the final statement, recall that finding the minimal suffix in $Suf[i,j]$ is one of the basic queries supported by the enhanced suffix array. □

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 S_j^ℓ is longer than $|S_j^{\ell-1}|$. For $\ell = 1$ we always set $B_j[1] = 1$, as S_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 the data structure takes $\mathcal{O}(n)$ space.

3.1 Queries

Assume we are looking for the minimal suffix of $T[i..j]$ with $\alpha(i,j) = \ell$. Our approach is based on Lemma 1. If case (a) holds, the lemma lets us compute the answer in $\mathcal{O}(1)$ time. In general we find the minimal suffix of S_j^ℓ, compare it with $T[p..j]$, and return the smaller of them.

We use both Lemma 1 and the bit vector B_j to compute the minimal suffix of S_j^ℓ. Let ℓ' be the largest index not exceeding ℓ such that $B_j[\ell'] = 1$. Note that such an index always exists (as $B_j[1] = 1$) and can be found in constant time using standard bit-wise operations. For any index $\ell'' \in \{\ell'+1, \ldots, \ell\}$ we have $B_j[\ell''] = 0$, i.e., case (b) of Lemma 1 holds for $S_j^{\ell''}$. This inductively implies that

the minimal suffix of S_j^ℓ is actually the minimal suffix of $S_j^{\ell'}$. On the other hand $B_j[\ell'] = 1$, so for the latter we have a guarantee that case (a) holds, which lets us find the minimal suffix of S_j^ℓ in constant time. This completes the description of an $\mathcal{O}(1)$-time query algorithm.

3.2 Construction

A simple $\mathcal{O}(n \log n)$-time construction algorithm also relies on Lemma 1. 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 S_j^ℓ for consecutive values of ℓ. Once we know the answer for $\ell - 1$, case (a) of Lemma 1 gives us the second candidate for the minimal suffix of S_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 not contained in $S_j^{\ell-1}$. Therefore we obtain the following result.

Theorem 2. *A string T of length n can be stored in an $\mathcal{O}(n)$-space structure that enables to compute the minimal suffix of any substring of T in $\mathcal{O}(1)$ time. This data structure can be constructed in $\mathcal{O}(n \log n)$ time.*

The construction described above is simple and works for any choice of canonical substrings, but, unfortunately, it cannot be used to achieve a trade-off between the query and the construction time. Below we consider a specific choice of canonical substrings and give an alternative construction method. The intuition behind such a choice is that given a string of length k we can compute the minimal suffix for each of its prefixes in $\mathcal{O}(k)$ total time. Hence it would be convenient to have many S_j^ℓ which are prefixes of each other. Then a natural choice is $\left|S_j^\ell\right| = 2^{\ell-1} + (j \bmod 2^{\ell-1})$, as then $S^\ell_{\alpha 2^{\ell-1}}, S^\ell_{\alpha 2^{\ell-1}+1}, \ldots, S^\ell_{(\alpha+1)2^{\ell-1}-1}$ are all prefixes of $S^\ell_{(\alpha+1)2^{\ell-1}-1}$. Unfortunately, such choice does not fulfill the condition $\left|S_j^\ell\right| \le 2\left|S_j^{\ell-1}\right|$, and we need to tweak it a little bit.

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

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

Note that if $2 \cdot 2^m \le j < 3 \cdot 2^m$, then $T[1..j] = S_j^{2m+2}$ while if $3 \cdot 2^m \le j < 4 \cdot 2^m$, then $T[1..j] = S_j^{2m+3}$. Clearly the number of such substrings ending at j is therefore $\mathcal{O}(\log n)$. The following facts show that the above choice of canonical substrings satisfies the remaining required properties.

Fact 3. *For any S_j^ℓ and $S_j^{\ell+1}$ with $\ell \ge 1$ we have $\left|S_j^{\ell+1}\right| < 2\left|S_j^\ell\right|$.*

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

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

while for odd ℓ

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

Fact 4. *For $1 \le i < j \le n$, the value $\alpha(i,j)$ can be computed in constant time.*

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

$$\left|S_j^{2m-1}\right| = 3 \cdot 2^{m-2} + (j \bmod 2^{m-2}) < 2^m \le |T[i..j]|$$
$$\left|S_j^{2m+2}\right| = 2 \cdot 2^m + (j \bmod 2^m) \ge 2^{m+1} > |T[i..j]|.$$

Thus $\alpha(i,j) \in \{2m-1, 2m, 2m+1\}$, and we can verify in constant time which of these values is the correct one. □

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 S_j^ℓ as follows. Fix $\ell > 1$ and split T into *chunks* of size 2^m each (with $m = \lfloor \ell/2 \rfloor - 1$). Now each S_j^ℓ is a prefix of a concatenation of at most four such chunks. Recall that given a string, a variant of Duval's algorithm (Algorithm 3.1 in [5]) takes linear time to compute the lengths of minimal suffixes of all its prefixes. We divide T into chunks of length 2^m (with $m = \lfloor \ell/2 \rfloor - 1$) and run this algorithm for each four (or less at the end) consecutive chunks. This gives the minimal suffixes of S_j^ℓ for all $1 \le j \le n$, in $\mathcal{O}(n)$ time. The value $B_j[\ell]$ is determined by comparing the length of the computed minimal suffix of S_j^ℓ with $|S_j^{\ell-1}|$. We have $\mathcal{O}(\log n)$ phases, which gives $\mathcal{O}(n \log n)$ total time complexity and $\mathcal{O}(n)$ total space consumption.

3.3 Trade-off

To obtain a data structure with $\mathcal{O}(n \log n/\tau)$ construction and $\mathcal{O}(\tau)$ query time, we define the bit vectors in a slightly different way. We set B_j to be of size $\lfloor \alpha(1,j)/\tau \rfloor$ with $B_j[k] = 1$ if and only if $k = 1$ or the minimal suffix of $S_j^{\tau k}$ is longer than $|S_j^{\tau(k-1)}|$. 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 $T[i..j]$ with $\alpha(i,j) = \ell$. As before, the difficult part is to find the minimal suffix of S_j^ℓ, and our goal is to find $\ell' \le \ell$ such that the minimal suffix of S_j^ℓ is actually the minimal suffix of $S_j^{\ell'}$, but is longer than $|S_j^{\ell'-1}|$. If $\ell = \tau k$ for an integer k, we could find the largest $k' \le k$ such that $B[k'] = 1$ and we would know that $\ell' \in (\tau(k'-1), \tau k']$. In the general case, we choose the largest k such that $\tau k \le \ell$, and then we know that we should consider $\ell' \in (\tau k, \ell]$ and $\ell' \in (\tau(k'-1), \tau k']$, 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 1 for $S_j^{\ell'}$ for one of these values. We simply generate all these candidates and use the enhanced suffix array to find the smallest suffix among them. In total, the query algorithm works in $\mathcal{O}(\tau)$ time, which gives the following result.

Theorem 5. *For any $1 \le \tau \le \log n$, a string T of length n can be stored in an $\mathcal{O}(n)$-space data structure that allows to compute the minimal suffix of any substring of T in $\mathcal{O}(\tau)$ time. This data structure can be constructed in $\mathcal{O}(n \log n/\tau)$ time.*

3.4 Applications

As a corollary we obtain an efficient data structure for computing Lyndon decompositions of substrings of T. A string w is a *Lyndon word* if it is strictly smaller than its proper cyclic rotations. For a nonempty 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 $w_1 > w_2 > \dots > w_k$ are Lyndon words [3]. The last factor w_k is the minimal suffix of x [5] and from the definition we easily obtain that $w_k^{\alpha_k}$ 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, which gives us the following corollary.

Corollary 6. *For any $1 \leq \tau \leq \log n$ a string T of length n can be stored in an $\mathcal{O}(n)$-space data structure that enables to compute the Lyndon decomposition of any 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. However, in contrast to that problem, the properties specific to maximal suffixes will let us design a linear time construction algorithm.

Observe that the only component of Section 3 which cannot be immediately adapted to the maximal suffix problem is Lemma 1. While its exact counterpart is not true, in Section 4.1 we prove the following lemma which is equivalent in terms of the algorithmic applications. Canonical substrings S_j^ℓ are defined exactly as before.

Lemma 7. *Consider a substring $T[i..j]$. Using the enhanced suffix array of T, one can compute in $\mathcal{O}(1)$ time an index p ($i \leq p \leq j$) such that the maximal suffix $T[\mu..j]$ of $T[i..j]$ is either equal to*

(a) $T[p..j]$, or
(b) the maximal suffix of $S_j^{\alpha(i,j)}$.

Just as the data structure described in Section 3, our data structure, apart from the enhanced suffix array, contains bit vectors B_j, $j \in [1, n]$, with $B_j[\ell] = 1$ if $\ell = 1$ or the maximal suffix of S_j^ℓ is longer than $|S_j^{\ell-1}|$. The query algorithm described in Section 3.1 can be adapted in an obvious way, i.e., so that it uses Lemma 7 instead of Lemma 1 and chooses the larger of the two candidates as the answer. This shows the following theorem:

Theorem 8. *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 3.2 and 3.3, also easily adapt to the maximal suffix problem. In this case, however, we can actually achieve $\mathcal{O}(n)$ construction time, as presented in Section 4.2.

4.1 Proof of Lemma 7

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 $S_j^{\alpha(i,j)}$ (case (b) of Lemma 7), the algorithm can return any $p \in [i, j]$. Below we assume that $T[\mu..j]$ is longer than $S_j^{\alpha(i,j)}$ and show that under this assumption the algorithm will return $p = \mu$. From the assumption and the properties of canonical substrings it follows that $\mu \in [i, r]$, where $r = j - |S_j^{\alpha(i,j)}|$, and that the lengths of the suffixes of $T[i..j]$ starting at positions in $[i, r]$ differ by up to a factor of two.

We start with locating the maximal suffix $T[p_1..]$ in $Suf[i, j]$. Then the maximal suffix $T[\mu..j]$ of $T[i..j]$ must start with $T[p_1..j]$, so $\mu \leq p_1$. To check if $\mu < p_1$, we locate the maximal suffix $T[p_2..]$ in $Suf[i, p_1 - 1]$. If $T[p_1..j]$ is not a prefix of $T[p_2..j]$, one can see that $\mu = p_1$. More formally, we have the following lemma, which appears as Lemma 2 in [1].

Lemma 9. *Let $P_1 = T[p_1..j]$ be a prefix of $T[\mu..j]$ and let $P_2 = T[p_2..j]$, where $T[p_2..]$ is the maximal suffix in $Suf[i, p_1 - 1]$. If P_1 is not a prefix of P_2, then $\mu = p_1$. Otherwise, P_2 is also a prefix of $T[\mu..j]$.*

We check if $P_1 = T[p_1..j]$ is a prefix of $P_2 = T[p_2..j]$. If not, we return p_1. If P_1 is a prefix of P_2, we know that $\mu \leq p_2$. To check if $\mu < p_2$ we could repeat the above step, i.e., locate the maximal suffix of $T[p_3..]$ in $Suf[i, p_2 - 1]$ and check if P_2 is a prefix of $P_3 = T[p_3..j]$. If not, we return $\mu = p_2$. If P_2 is a prefix of P_3, we again repeat the whole step. Unfortunately the number of repetitions could then be very large. Therefore we use the property that $2|P_1| \geq |P_2|$ to quickly jump to the last repetition. Informally, we apply the periodicity lemma to show that the situation must look like the one in Fig. 1. We prove this in two lemmas, which are essentially Lemmas 4 and 5 of [1]. We give their proofs here because we use different notation.

Lemma 10. *The substring $\rho = T[p_2..p_1 - 1]$ is the shortest period of P_2, i.e., ρ is the shortest string such that for some $s \geq 1$ one has $P_2 = \rho^s \rho'$.*

Proof. Since P_1 is a border of P_2, $\rho = T[p_2..p_1 - 1]$ is a period of P_2. It remains to prove that no shorter period is possible. So, consider the shortest period γ, and assume that $|\gamma| < |\rho|$. Then $|\gamma| + |\rho| \leq 2|\rho| \leq |T[p_2..j]|$, and by the periodicity lemma substring P_2 has another period $\gcd(|\gamma|, |\rho|)$. Since γ is the shortest period, $|\rho|$ must be a multiple of $|\gamma|$, i.e., $\rho = \gamma^k$ for some $k \geq 2$.

Suppose that $T[p_1..] \prec \gamma T[p_1..]$. Then 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 $1 \leq \ell \leq k$, so from the transitivity of $\prec$ we get that $T[p_1..] \prec \gamma^k T[p_1..] = T[p_2..]$, which contradicts the maximality of $T[p_1..]$ in $Suf[i, r]$. Therefore $T[p_1..] \succ \gamma T[p_1..]$, and consequently $\gamma^{k-1} T[p_1..] \succ \gamma^k T[p_1..]$. 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 $Suf[i, p_1 - 1]$, a contradiction. □

Lemma 11. *Suppose that $P_2 = \rho P_1 = \rho^s \rho'$. The maximal suffix $T[\mu..j]$ is the longest suffix of $T[i..j]$ equal to $\rho^t \rho'$ for some integer t. (See also Fig. 1.)*

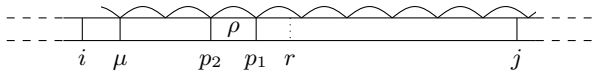

Fig. 1. A schematic illustration of Lemma 11.

Proof. Clearly P_2 is a border of $T[\mu..j]$. From $P_2 = \rho P_1$ and $|T[\mu..j]| \leq 2|P_1|$ we have $|T[\mu..j]| + |\rho| \leq 2|P_1| + |\rho| \leq 2|P_2|$. Consequently the occurrences of P_2 as a prefix and as a suffix of $T[\mu..j]$ have an overlap with at least $|\rho|$ positions. As $|\rho|$ is a period of P_2, this implies that $|\rho|$ 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. Moreover ρ^2 is a prefix of $T[\mu..j]$, since it is a prefix of P_2, which is a prefix of $T[\mu..j]$. Now $\rho'' \neq \varepsilon$ would imply a non-trivial occurrence of ρ in ρ^2, which contradicts ρ being primitive, see [4]. Thus $T[\mu..j] = \rho^r\rho'$. If $t > r$, then $\rho^t\rho' \succ \rho^r\rho'$, so $T[\mu..j]$ is the longest suffix of $T[i..j]$ equal to $\rho^t\rho'$ for some integer t. □

Proof (of Lemma 7). Let $T[p_1..]$ be the maximal suffix in $Suf[i, r]$ and $T[p_2..]$ be the maximal suffix in $Suf[i, p_1 - 1]$. We first compute p_1 and p_2 in constant time using the enhanced suffix array. Then we check if $T[p_1..j]$ is a prefix of $T[p_2..j]$. If it is not, we return $p = p_1$. Otherwise we compute the largest integer r such that ρ^r (for $\rho = T[p_2..p_1 - 1]$), is a suffix of $T[i..p_1 - 1]$ using the method described in Section 2, and return $p = p_1 - r|\rho|$. From the lemmas above it follows that if $T[\mu..j]$ is longer than $S_j^{\alpha(i,j)}$, then $p = \mu$. □

4.2 Construction

Our algorithm is based on the following notion. For $1 \leq p \leq j \leq n$ we say that a position p is *j-active* if there is no position $p' \in [p + 1, j]$ such that $T[p..j] \prec T[p'..j]$. Equivalently, p is j-active exactly when $T[p..j]$ is its own maximal suffix. The maximal suffix of any string is its own maximal suffix, so from the definition it follows that the starting position of the maximal suffix of $T[i..j]$ is the minimum j-active position in $[i, j]$. Therefore, for $\ell > 1$ we have $B_j[\ell] = 1$ if and only if there is at least one j-active position within the range $R_j^\ell = \left[j - |S_j^\ell| + 1, j - |S_j^{\ell-1}|\right]$. We set $R_j^1 = [j, j]$ so that this equivalence also holds for $\ell = 1$ (since j is always j-active).

Example 12. If $T[1..8] = \texttt{dcccabab}$, the 8-active positions are $1, 2, 3, 4, 6, 8$.

The construction algorithm iterates over j ranging from 1 to 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 3.2, 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 13. *If the list of all $(j-1)$-active positions consists of $p_1 < p_2 < \ldots < p_z$, 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 $T[j + p_k - p_{k+1}] < T[j]$, remove p_k from the list.*

Proof. First we prove 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 $(j-1)$-active positions and j.

Secondly, note that if $1 \le p \le j-1$ is a $(j-1)$-active position and $T[p'..j-1]$ is a prefix of $T[p..j-1]$, then p' is $(j-1)$-active too. If not, then there exists a position p'', $p' < p'' < j-1$, such that $T[p'..j-1] \prec T[p''..j-1]$, and it follows that $T[p..j-1] = T[p'..j-1]T[j+p-p'..j-1] \prec T[p''..j-1]$, a contradiction.

A $(j-1)$-active position p is not j-active only if (1) $T[p] < T[j]$ or (2) there exists $p < p' \le j-1$ such that $T[p'..j-1]$ is a prefix of $T[p..j-1]$, i.e., p' is $(j-1)$-active, and $T[p'..j] \succ T[p..j]$, or, equivalently, $T[j+p-p'] < T[j]$. Both of these cases are detected by the deletion procedure. □

Example 14. If $T[1..9] =$ `dcccababb`, the 8-active positions are $1, 2, 3, 4, 6, 8$, and the 9-active positions are $1, 2, 3, 4, 8, 9$, i.e., we add 9 and delete 6.

Fact 15. *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. □

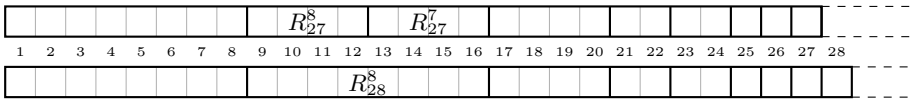

Fig. 2. The partitions of $[1, j]$ into R_j^ℓ for $j = 27$ and $j = 28$. As for $j = 28$ we have $k = 2$ and $2k+4 = 8$, R_{27}^7 and R_{27}^8 are merged into R_{28}^8.

We scan the positions of T from left to right computing the bit vectors. We maintain 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 $(j-1)$-active position and turn it into the list of j-active positions, we also maintain for every j' a list of pointers to pairs of neighbouring positions. Whenever a new pair of neighbouring positions p_k, p_{k+1} appears, we compute $L = \mathrm{lcp}(T[p_k..], T[p_{k+1}..])$, and insert a pointer to the pair p_k, p_{k+1} into the list of a position $j' = p_{k+1} + L$. When we actually reach $j = j'$, we follow the pointer and 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. From Lemma 13 it follows that the two possible updates of the list under transition from $j-1$ to j are either adding j or deleting some position from the list. This guarantees that the process of deletion from Lemma 13 and the process we have just described are equivalent.

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. First we update the list of $(j-1)$-active positions. When a position is deleted from the list, we find the range it belongs to, and decrement the counter of active positions there. If a counter drops down to zero, we clear the corresponding bit of the bit vector. Then we start updating the partition: first we append a new range $[j, j]$ to the partition of $[1..j-1]$ and initialize the counter of active positions there to one. Then, k being the largest power of 2 dividing j, we update the first $2k+4$ ranges using Fact 15, 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.

Theorem 16. *A string T of length n can be stored in an $\mathcal{O}(n)$-space structure that allows computing the maximal suffix of any substring of T in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

References

1. M. Babenko, I. Kolesnichenko, and T. Starikovskaya. On minimal and maximal suffixes of a substring. In J. Fischer and P. Sanders, editors, *CPM*, volume 7922 of *Lecture Notes in Computer Science*, pages 28–37. Springer, 2013.
2. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
3. K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *The Annals of Mathematics*, 68(1):81–95, 1958.
4. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
5. J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
6. S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.