# Computing the Longest Unbordered Substring

Pawel Gawrychowski[1,⋆], Gregory Kucherov[2], Benjamin Sach[3], and Tatiana Starikovskaya[3]

[1] Warsaw Center of Mathematics and Computer Science
[2] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS
[3] University of Bristol

**Abstract.** A substring of a string is *unbordered* if its only border is the empty string. The study of unbordered substrings goes back to the paper of Ehrenfeucht and Silberger [Discr. Math 26 (1979)]. The main focus of their and subsequent papers was to elucidate the relationship between the longest unbordered substring and the minimal period of strings. In this paper, we consider the algorithmic problem of computing the longest unbordered substring of a string. The problem was introduced recently by G. Kucherov et al. [CPM (2015)], where the authors showed that the average-case running time of the simple, border-array based algorithm can be bounded by $\mathcal{O}(\max\{n, n^2/\sigma^4\})$ for $\sigma$ being the size of the alphabet. (The worst-case running time remained $\mathcal{O}(n^2)$.) Here we propose two algorithms, both presenting substantial theoretical improvements to the result of [11]. The first algorithm has $\mathcal{O}(n \log n)$ average-case running time and $\mathcal{O}(n^2)$ worst-case running time, and the second algorithm has $\mathcal{O}(n^{1.5})$ worst-case running time.

## 1 Introduction

A proper prefix of a string that is simultaneously its suffix is called a *border*. If the only border of a substring is the empty string, then this substring is called *unbordered*. The study of unbordered substrings commenced in the 1979 paper of Ehrenfeucht and Silberger [6]. The main focus of [6] and of subsequent papers [1, 4, 8] was to clarify the relationship between the maximal length of an unbordered substring of a string and its periodicity. As a result of this line of research, it was shown that in order to guarantee the equality between the maximal length of an unbordered substring and the minimal period, either the former should be smaller than 3/7 of the string length, or the latter should be smaller than 1/2 of the string length, where both bounds are tight. In this work, we focus on the computational problem that can be considered complementary to the previous study: Given a string $T$ of length $n$, compute its unbordered substring of maximal length.

It is well-known that the minimal period can be easily computed on $\mathcal{O}(n)$ time by a variant of the Knuth-Morris-Pratt algorithm [12]. Note that if a string

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

is periodic, i.e. its minimal period is at most half of the string length, then a longest unbordered substring can be found as an unbordered conjugate of string's root (which is a substring of minimal period length). This computation can be done in $\mathcal{O}(n)$ time as well [5].

However, this approach is not applicable in the general case. It can be easily seen that an unbordered substring cannot be longer than the minimal period of the string (as any substring longer than the period has a border). For most strings, the maximal length of an unbordered substring, and consequently their minimal period, are large. (More formally, it was shown recently that the average maximal length of an unbordered substring of a string of length $n$ is at least $(1 - 8/\sigma^4)\ n$ for alphabets of size $\sigma$ [11].) In this case, it is no longer possible to exploit the relation between unbordered substrings and the minimal period. A straightforward way to compute the longest unbordered substring is to compute the border array of each suffix of the string [12]. This algorithm has quadratic worst-case running time, and no better worst-case bound has been obtained, to the best of our knowledge. In [12], it was shown that the average-case running time of this algorithm is $\mathcal{O}(\max\{n, n^2/\sigma^4\})$. The average-case time complexity captures the 'typical' running time of the algorithm, rather than the running time on most hard problem instances [13]. Other problems on strings studied under this model include pattern matching, edit distance, suffix trees, and more (see e.g. [9]).

We give two algorithms for computing an unbordered substring of the maximal length: algorithm $\mathcal{A}$ and algorithm $\mathcal{B}$. Both algorithms have $\mathcal{O}(n)$ space complexity. The worst-case running time of algorithm $\mathcal{A}$ is $\mathcal{O}(n^2)$ – the same as of the simple, border-array based algorithm. However, its average-case time complexity, i.e. the time complexity averaged over all input strings of length $n$, is $\mathcal{O}(n \log n)$ which provides a considerable improvement to the bound of [11]. For algorithm $\mathcal{B}$, we show an $\mathcal{O}(n^{1.5})$ *worst-case* time bound. To our knowledge, this is the first sub-quadratic worst-case bound for this problem. We assume the word-RAM model of computation with $\Omega(\log n)$-bit words and an integer alphabet of polynomial size in $n$.

Both algorithms distinguish between two types of substrings that have a non-empty border: those having a 'short' border (shorter than a threshold $\tau$) and those having only 'long' borders (longer than $\tau$). For each position $j$, there are only $\tau$ possible short borders, which allows to identify the substrings $T[i..j]$ that have short borders quickly. On the other hand, we will show that the number of substrings $T[i..j]$ that have only long borders is small, which will also make it possible to identify them quickly.

## 2 Preliminaries

Let $\Sigma$ be a finite *alphabet*. The elements of $\Sigma$ 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* $n$ consists of letters $T[1], T[2], \dots, T[n]$. The length $n$ of $T$ is denoted by $|T|$. For $1 \leq i \leq j \leq n$,

$T[i..j]$ is a *substring* of $T$ with endpoints $i$ and $j$. A substring $T[1..j]$ is called a *prefix* of $T$, and a substring $T[i..n]$ is called a *suffix* of $T$. A prefix (or a suffix) of $T$ different from $T$ is called *proper*.

## 2.1 Borders and periods.

If a proper prefix of a string is simultaneously its suffix, then it is called a *border*. A string is called *unbordered* if the only border it has is the empty string. We define the *border array* $B$ of $T$ to contain the lengths of the longest borders of all prefixes of $T$, i.e. $B[i]$ is the length of the longest border of $T[1..i]$, $i = 1..n$. The last entry in the border array, $B[n]$, contains the length of the longest border of $T$. It is well-known that the border array and therefore the longest border of $T$ can be computed in $\mathcal{O}(n)$ time and space [12].

We remark that the border array construction algorithm immediately gives an $\mathcal{O}(n^2)$-time algorithm for computing the longest unbordered substring of $T$: It suffices to build the border arrays of all suffixes of $T$. Then the longest unbordered substring starting at position $i$ will correspond to the rightmost entry in the border array of $T[i..n]$ containing zero.

A *period* of $T$ is a positive integer $\pi$ such that for all $i$, $1 \leq i \leq n - \pi$, $T[i] = T[i + \pi]$. The smallest of all periods of $T$ is called the *minimal* period of $T$. The minimal period of $T$ is equal to $n - B[n]$, and hence can be computed in $\mathcal{O}(n)$ time. Note that if $T$ is unbordered, then its smallest period is equal to its length. Note also that a border of a border of a string is again a border of that string, and that the shortest border is unbordered.

We will also exploit the Periodicity lemma:

**Lemma 1.** *If a string $T$ has periods $\rho$ and $\gamma$ such that $\rho + \gamma \leq |T|$, then $T$ has period $\gcd(\rho, \gamma)$, the greatest common divisor of $\rho$ and $\gamma$.*

Finally, we will make use of the *shortest border array* $B'$ of $T$ which is defined to contain the lengths of the shortest borders of all prefixes of $T$. That is, for each $i = 1..n$, $B'[i]$ is the length of the shortest non-empty border of $T[1..i]$ if $T[1..i]$ has a non-empty border, and zero otherwise. It is not difficult to see that the shortest border array of $T$ can be computed in linear time. It suffices to run the standard border array construction algorithm, then if $B[i]$ is the longest border of $T[1..i]$, the shortest border of $T[1..i]$ equals $B'[B[i]]$ and can be computed in $\mathcal{O}(1)$ time.

## 2.2 Suffix trees and auxiliary data structures.

The (generalized) suffix tree of a set $S$ of strings is a compacted trie of suffixes of the strings in $S$, where the suffixes of the $i$-th string are appended with a special letter $\$_i$ that does not belong to the alphabet $\Sigma$ [14]. In this paper, we will consider the suffix trees for the following sets of strings:

- a singleton set containing $T$,

– a singleton set containing the reverse of $T$,
– a two-element set consisting of $T$ and some substring $S$ of $T$.

We assume that we know string depths of all branching nodes. The string depth of a node is the length of the string formed by the labels from the root to that node. We also assume that we have access to the corresponding suffix array for each suffix tree. The $j$-th entry in the suffix array gives the position $i$ of the start of the $j$-th largest suffix in lexicographical order. We assume that each entry in the suffix array holds a pointer to the corresponding leaf in the suffix tree and vice versa. Furthermore we assume each internal node in the suffix tree holds a pointer to the leftmost and rightmost leaves in each subtree. We remark that the suffix array is not strictly required to achieve the claimed bounds in either algorithm but it will simplify the explanation.

As is relatively standard, we augment each tree with the *lowest common ancestor* (LCA) and the *range minimum query* (RMQ) data structures [2]. The RMQ data structure is built on top of the suffix array for the corresponding tree. We omit the definitions as these data structures are used only indirectly via Lemmas 2 and 3 below. On alphabets of size $|T|^{\mathcal{O}(1)}$ all suffix trees, as well as the suffix arrays, the LCA and the RMQ data structures, can be constructed in $\mathcal{O}(|T|)$ time and occupy $\mathcal{O}(|T|)$ space [7, 2]. Augmented in this way, the suffix trees become a very powerful tool:

**Lemma 2.** *Using the augmented suffix trees/arrays of $T$ and the reverse of $T$, the following queries can be answered in $\mathcal{O}(1)$ time:*

1. *Given endpoints of two substrings $S_1$ and $S_2$ of $T$, decide whether $S_1 = S_2$,*
2. *Given an interval in the suffix array of $T$ find the suffix in the interval with the smallest starting position,*
3. *Given endpoints of two substrings $S_1, S_2$ of $T$, compute the longest common suffix of $S_1$ and $S_2$,*
4. *Given endpoints of two substrings $S_1, S_2$ of $T$ compute the largest integer $\alpha$ and the longest suffix $S$ of $S_1$ such that $SS_1^\alpha$ is a suffix of $S_2$. Here $S_1^\alpha$ denotes the string formed by $\alpha$ repetitions of $S_1$.*

**Lemma 3.** *Using the suffix tree of $T$ and the suffix tree of $T$ and some substring $S$ of $T$, the following queries can be answered in $\mathcal{O}(|T|)$ time:*

1. *Retrieve all suffixes of $T$ that are not prefixes of other suffixes of $T$, sorted in lexicographic order,*
2. *For each suffix $T[i..n]$ of $T$, compute the length of its longest prefix $P_i$ that occurs in $S$ and the first position of such an occurrence.*

Lemmas 2 and 3 are proved using standard suffix tree algorithms, perhaps with the only exception of Query 4 of Lemma 2. We answer this query in the following way. First, we find the longest common suffix of $S_1$ and $S_2$ (Query 3 of Lemma 2). If its length is smaller than $|S_1|$, we set $\alpha$ to zero and $S$ to the suffix. Otherwise, $S_1$ is a suffix of $S_2$. Let $S_2 = T[i..j]$. Then $SS_1^{\alpha-1}$ is equal to the longest common suffix of $T[i..j]$ and $T[1..j - |S_1| + 1]$ and can be found in $\mathcal{O}(1)$ time by one more Query 3 of Lemma 2.

## 3 Algorithm $\mathcal{A}$

In this section we describe algorithm $\mathcal{A}$, which has $\mathcal{O}(n^2)$ *worst-case* time complexity and $\mathcal{O}(n \log n)$ *average-case* time complexity. Recall from the introduction, that both our algorithms set a threshold to distinguish between short and long borders. For algorithm $\mathcal{A}$, we set the threshold, $\tau$ to $6 \log n$[1]. That is, a non-empty border is short if its length is smaller than $6 \log n$, and long otherwise. We start with the following lemma which says that strings containing substrings with long borders are very rare. We can therefore afford to process them less efficiently and still achieve a good average-case time complexity.

**Lemma 4.** *Consider a random string $T$ of length $n$ with i.i.d. distribution of letters over a non-unary alphabet. The probability that $T$ contains a substring with a long border is smaller than $\frac{1}{n}$.*

*Proof.* If $T$ contains a substring with a long border, then there is a substring $T[i..j]$ with a border of length $6 \log n$ and consequently $T$ contains a pair of equal substrings of length $6 \log n$. Furthermore, either these two equal substrings do not overlap, or they can be shortened to produce two non-overlapping substrings of length $3 \log n$, or the length of $T[i..j]$ is at most $9 \log n$. In the last case, the minimal period of $T[i..j]$ is at most $9 \log n - 6 \log n = 3 \log n$, and the prefix of length $3 \log n$ of $T[i..j]$ and a substring of the same length starting with the next full repetition of the period do not overlap. So in all the cases, there exist two non-overlapping equal substrings of length $3 \log n$.

Now we will show that the probability of a random string to have two non-overlapping equal substrings of length $3 \log n$ is small. Consider any two such substrings. Since they do not overlap and their letters are chosen uniformly and independently, the probability of the substrings being equal is at most $1/n^3$ (recall that the alphabet cardinality is at least 2). Since there are at most $n^2$ pairs of substrings of length $3 \log n$, by the union bound the probability of at least one such pair being equal is at most $1/n$. $\square$

The string, $T$ contains a substring with a long border if and only if the suffix tree of $T$ contains a branching node with string depth at least $6 \log n$. We can check whether this is true in $\mathcal{O}(n)$ time. If it is, we run the simple $\mathcal{O}(n^2)$-time algorithm to compute the longest unbordered substring. We can now proceed under the assumption that $T$ contains no substring with a long border.

Algorithm $\mathcal{A}$ considers each position in the string $T$ in turn and determines the largest unbordered substring that ends at that position. Consider an arbitrary position $j$ and a substring $T[i..j]$. If $T[i..j]$ has a border, the border must be short and hence equal to one of $T[j - 6 \log n + 1..j], T[j - 6 \log n + 2..j], \ldots, T[j]$. Remember that our objective is to compute the smallest position $i$ such that $T[i..j]$ is unbordered. It follows that we need to compute the smallest position $i$ with no occurrence of $T[j - 6 \log n + 1..j], T[j - 6 \log n + 2..j], \ldots, T[j]$.

[1] We assume the logarithm base to be 2 throughout the paper.

Occurrences of any substring of $T$ form an interval in the suffix array. This property is immediate from the lexicographical ordering. The positions where none of these substrings occur correspond exactly to the complement of these $\mathcal{O}(\log n)$ intervals. It therefore follows that these 'complementary' positions also form $\mathcal{O}(\log n)$ disjoint intervals in the suffix array. We can find these 'complementary' intervals by sorting the original intervals. The *smallest* position $i$ where none of these substrings occur is the minimum position in any of the complementary intervals. We can compute the minimum in $\mathcal{O}(\log n)$ time by performing Query 2 on each of the intervals in $\mathcal{O}(1)$ time and reporting the minimum.

To find the intervals efficiently, we use the following lemma to retrieve the *locus* (the node labelled by the substring) of each substring $T[j - 6\log n + 1..j], T[j - 6\log n + 2..j], \ldots, T[j]$ in the suffix tree of $T$ in constant time per substring. If the substring occurs only implicitly in the suffix tree, the locus is the node at the deeper end of the edge where the substring ends. We can then determine the suffix array interval corresponding to a locus in $\mathcal{O}(1)$ time by following the pointers to the leftmost and rightmost leaves in the subtree rooted at the locus and then following the pointers to the corresponding suffix array locations.

**Lemma 5.** *The suffix tree of $T$ can be preprocessed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, so that the locus of any $T[j-\ell..j]$ such that $0 \le \ell < 6\log n$ can be retrieved in constant time.*

*Proof.* For every leaf corresponding to a suffix $T[i..n]$ we store a bitvector of length $6\log n$, where the $\ell$-th bit is set to $\mathbf{1}$ iff the leaf has an ancestor at string depth $0 \le \ell < 6\log n$. The bitvectors can be constructed in $\mathcal{O}(n\log n)$ time in a straightforward manner and occupy $\mathcal{O}(n)$ (words of) space.

For each bitvector we build the rank/select data structure [10]. The data structures can be built in $\mathcal{O}(n\log n)$ time, occupy $\mathcal{O}(n)$ (words of) space, and allow to compute the number $m$ of $\mathbf{1}$ in a prefix of a bitvector of given length $\ell$ and to find the $m$-th $\mathbf{1}$ in a bitvector $\mathcal{O}(1)$ time.

Next, we augment the suffix tree with the *level ancestor* data structure in $\mathcal{O}(n)$ time and space that given an integer $d$ and a node allows to compute the node's ancestor of (node) depth $d$ in $\mathcal{O}(1)$ time [3].

To retrieve the locus of $T[j - \ell..j]$ we use the rank/select data structure of the bitvector stored for $T[j - \ell..n]$ to compute the number $m$ of ancestors of the corresponding leaf of string depths less than $\ell$ in constant time. The locus of $T[j - \ell..j]$ is the ancestor of the leaf of depth $d = m + 1$, and can be retrieved in $\mathcal{O}(1)$ time. □

To sort the intervals efficiently, we build the suffix tree (and the suffix array) of the string $T_j = T[j-6\log n+1..j]$. This gives us the lexicographical ordering of the substrings $T[j-6\log n+1..j], T[j-6\log n+2..j], \ldots, T[j]$. We can use this to sort the corresponding intervals as follows. First we remove any substring $T[j-\ell_1+1..j]$ which has another substring $T[j-\ell_2+1..j]$ as a prefix. This cannot affect correctness as the interval in the suffix array for $T$ corresponding to occurrences of $T[j - \ell_1 + 1..j]$ is completely contained within the interval corresponding to

$T[j - \ell_2 + 1..j]$. These substrings can be removed in $\mathcal{O}(\log n)$ time by applying Query 1 of Lemma 3. Finally, the key observation is that the remaining intervals do not intersect and the order of the remaining intervals within the suffix array of $T$ corresponds to the lexicographical order of the corresponding substrings. This gives the desired $\mathcal{O}(\log n)$ time to compute the longest unbordered substring ending at a single position $j$.

**Theorem 1.** *The worst-case time complexity of algorithm $\mathcal{A}$ is $\mathcal{O}(n^2)$ and the average-case time complexity is $\mathcal{O}(n \log n)$. The space complexity of the algorithm is $\mathcal{O}(n)$.*

*Proof.* It is easy to see the bounds on the *worst-case* time complexity and the space complexity of the algorithm. We now show the average-case time complexity. If $t$ is the running time in the case when there are no long borders (i.e. no branching nodes of string depth $\geq 6 \log n$ in the suffix tree of $T$), then by Lemma 4 the average-case time complexity is bounded by $\mathcal{O}(\frac{1}{n} \cdot n^2) + 1 \cdot t = \mathcal{O}(n) + t$.

It remains to show that $t = \mathcal{O}(n \log n)$. We start by building the suffix tree of $T$ and preprocessing it according to Lemma 5 in $\mathcal{O}(n \log n)$ time. Then, for each $j = 1..n$ we build the suffix array and suffix tree of $T_j$ and retrieve its suffixes that are not prefixes of other suffixes in $\mathcal{O}(\log n)$ time. For each of the retrieved suffixes we compute the interval of its occurrences. We then sort these intervals and take the complement of the intervals in $\mathcal{O}(\log n)$ time. The complement is a union of at most $6 \log n$ disjoint intervals and we compute the minimum in these intervals in $\mathcal{O}(\log n)$ time. The claim follows. $\square$

We remark that in the case of large alphabets (of size $n^{\Omega(1)}$) it is impossible to use the algorithm [7] to build the suffix trees of substrings $T[j - 6 \log n + 1..j]$ in linear time. To overcome this technicality, we apply the following alphabet reduction trick prior to constructing the trees. We partition $T$ into $\mathcal{O}(n)$ blocks of length $12 \log n$ with overlaps of $6 \log n$ positions. We sort letters in each block and for each letter $T[i]$ store its rank in the block. Each $T[j - 6 \log n + 1..j]$ belongs to at least one of the blocks. To construct its suffix tree, we consider one of the blocks containing it and replace all letters with their ranks in the block. This reduces the size of the alphabet to $\mathcal{O}(\log n)$ and makes it possible to use the algorithm [7]. The alphabet reduction trick takes $\mathcal{O}(n \log n)$ extra time and does not affect the time complexity of the algorithm.

## 4 Algorithm $\mathcal{B}$

In this section we describe algorithm $\mathcal{B}$, which has $\mathcal{O}(n^{1.5})$ *worst-case* time complexity. As in Algorithm $\mathcal{A}$, we set a threshold to distinguish between short and long borders. For algorithm $\mathcal{B}$, we set the threshold, $\tau$ to $\sqrt{n}$. That is, a non-empty border is short if its length is smaller than $\sqrt{n}$, and long otherwise.

First note that we can compute the longest unbordered substring of length at most $4\sqrt{n}$ in $\mathcal{O}(n^{1.5})$ time by computing the border array of each substring of length $4\sqrt{n}$. From now on, we are only interested in unbordered substrings of length at least $4\sqrt{n}$. The algorithm will consist of $\sqrt{n}$ stages. At stage $k$ it computes the longest unbordered substring that ends in an interval $J_k = [k\sqrt{n}+1, (k+1)\sqrt{n}]$. Let $F_k^i$, $i = 1..(k-3)\sqrt{n}$, be the set substrings of $T$ that start at position $i$ and end in $J_k$. The algorithm considers each $i = 1..(k-3)\sqrt{n}$ in order and either says that there is no unbordered substring in $F_k^i$ or retrieves a substring $T[i..j] \in F_k^i$. We guarantee that $T[i..j]$ does not have short borders. Furthermore, if there are unbordered substrings in $F_k^i$, we guarantee that $T[i..j]$ is the longest of them. We refer to $T[i..j]$ as the candidate. After retrieving the candidate $T[i..j]$, the algorithm checks if it is unbordered. If it is, the algorithm updates the maximal length of unbordered substrings.

## 4.1 Candidates

Let $P_i$ be the longest prefix of $T[i..n]$ that occurs in $T_k = T[(k-1)\sqrt{n}+1..(k+1)\sqrt{n}]$. If $T[i..j] \in F_k^i$ has a short border, then this border is a prefix of $P_i$. Moreover, if $\ell$ is the position of an occurrence of $P_i$ in $T_k$ and $\ell+|P_i|-1 < j$, then $T[\ell..j]$ has a non-empty border of length at most $|P_i|$. This simple observation will allow us to differentiate between substrings with short borders and without those. We explain the technical details below.

*Preprocessing.* We start by constructing the suffix tree for two strings $T$ and $T_k$. With its help we compute, for each $i = 1..(k-3)\sqrt{n}$, the length and the position of an occurrence of the longest prefix $P_i$ of $T[i..n]$ that occurs in $T_k$ (Query 2 of Lemma 3).

Consider all conjugates $T[\ell..(k+1)\sqrt{n}]]$ \$ $T[(k-1)\sqrt{n}+1..\ell-1]$ of $T_k$\$, where \$ is a letter that does not belong to the main alphabet. We compute the shortest border array for each of the conjugates in $\mathcal{O}(n)$ time in total. Obviously, values in the arrays are bounded by $2\sqrt{n}$. We sort each array in $\mathcal{O}(\sqrt{n})$ time using bucket sort. Overall, it takes $\mathcal{O}(n)$ space and time. For $r \in [k\sqrt{n}+1, (k+1)\sqrt{n}]$ let

$$S_{\ell,r} = \begin{cases} T[\ell..r], \text{ if } r > \ell; \\ T[\ell..(k+1)\sqrt{n}] \,\$\, T[(k-1)\sqrt{n}+1..r], \text{ if } r < \ell. \end{cases}$$

We define $r_\ell^p$ to be the largest position in $J_k \setminus [\ell, \ell+p-1]$ such that $S_{\ell,r_\ell^p}$ is either unbordered or has the shortest border of length at least $p+1$. For a fixed $\ell$, all values $r_\ell^p$ can be computed in $\mathcal{O}(\sqrt{n})$ time by scanning the (sorted) shortest border array for $T[\ell..(k+1)\sqrt{n}]]$ \$ $T[(k-1)\sqrt{n}+1..\ell-1]$.

*Computing candidates.* Below we fix $i$ and show how to compute the candidate in $F_k^i$. If $P_i$ is the empty string, $T[i..(k+1)\sqrt{n}]$ is the longest, unbordered substring in $F_k^i$ and we return it as the candidate. Otherwise, let $\ell$ be the position of an occurrence of $P_i$ in $T_k$ and let $p = |P_i|$.

**Lemma 6.** *If $r_\ell^p$ is not defined, then $F_k^i$ contains no unbordered substrings.*

*Proof.* It suffices to show that $T[i..j]$ ends with a prefix of $P_i$ for all $j \in J_k$. If $j \in [\ell, \ell+p-1]$, the claim is obvious. Consider now $j \in J_k \setminus [\ell, \ell+p-1]$. We know that $S_{\ell,j}$ has a border of length in $[1,p]$, which means that $S_{\ell,j}$ ends with a prefix of $P_i$. Since $S_{\ell,j}$ is a suffix of $T[i..j]$, we obtain that $T[i..j]$ also ends with a prefix of $P_i$. □

If the condition of the lemma is satisfied, the algorithm says that $F_k^i$ contains no unbordered substrings. Otherwise, let $j = r_\ell^p$. Note that $|S_{\ell,j}| \geq p+1$ by definition.

**Lemma 7.** *$T[i..j]$ does not have a short border.*

*Proof.* The proof is by contradiction. Suppose that $T[i..j]$ has a short border $B$. As $B$ is a prefix of $T[i..n]$ and occurs in $T_k$, it must be a prefix of $P_i$ (not necessarily proper). Consequently, $S_{\ell,j}$ starts with $B$ and ends with $B$. Hence, $B$ is a border of $S_{\ell,j}$ of length $|B| \in [1,p]$, which contradicts the definition of $j$. □

**Lemma 8.** *If $F_k^i$ contains unbordered substrings, then $T[i..j]$ is the longest of them.*

*Proof.* Let us first show that if a substring $T[i..j'] \in F_k^i$ is unbordered, then $S_{\ell,j'}$ is either unbordered or has the shortest non-empty border of length at least $p+1$. Suppose that the shortest non-empty border of $S_{\ell,j'}$ has length in $[1,p]$. This border is a prefix of $P_i$, i.e. $S_{\ell,j'}$ ends with a prefix of $P_i$. Consequently, $T[i..j']$ is not unbordered as it starts with $P_i$ and ends with the prefix of $P_i$, a contradiction.

It follows that all unbordered substrings in $F_k^i$ have length at most $|T[i..j]|$. It remains to show that if $T[i..j]$ has a long border, then all shorter substrings in $F_k^i$ have a non-empty border. As $T[i..j]$ has a long border, for some $b \geq \sqrt{n}$ we have $T[i..i+b-1] = T[j-b+1..j]$. It follows that for all $j' \in [k\sqrt{n}+1, j]$ we have $T[i..i+b+j-j'-1] = T[j-b+1..j']$, i.e. all substrings $T[i..j']$ shorter than $T[i..j]$ have a non-empty border. □

The algorithm retrieves the candidate $T[i..j]$ in $\mathcal{O}(1)$ time. The last two lemmas guarantee that $T[i..j]$ does not have short borders and if there are unbordered substrings in $F_k^i$, then $T[i..j]$ is the longest of them. It remains to check if $T[i..j]$ is unbordered. As it has no short borders, it suffices to check if it has a long border.

## 4.2 Long border check

Let $S_j$ be the shortest suffix of $T[1..j]$ such that its minimal period is larger than $\sqrt{n}/2$. We will show that every long border of $T[i..j]$ ends with an occurrence of $S_j$. During the long border check we will scan over a sorted list of occurrences of $S_j$ to determine if one of them induces a long border of $T[i..j]$.

*Preprocessing.* Let $S_j$ be the shortest suffix of $T[1..j]$ such that its minimal period is larger than $\sqrt{n}/2$. If there is no such suffix, $S_j$ is undefined.

**Lemma 9.** *If $T[i..j]$ is unbordered or has only long borders, then $S_j$ is defined and all long borders of $T[i..j]$ (if any) end with an occurrence of $S_j$.*

*Proof.* If $T[i..j]$ is unbordered, then its shortest period is equal to its length which is larger than $\sqrt{n}$. This shows that $S_j$ is defined.

If $T[i..j]$ has only long borders, then consider the shortest of them. Since the shortest border is always unbordered, its shortest period is equal to its length which is larger than $\sqrt{n}$. This implies that $S_j$ can only be shorter than this shortest long border. Thus, $S_j$ is a suffix of the shortest long border, and therefore a suffix of any long border. □

**Lemma 10.** *For any $j$ there are at most $2\sqrt{n}$ occurrences of $S_j$ in $T$.*

*Proof.* Since the minimal period of $S_j$ is larger than $\sqrt{n}/2$, any two occurrences of $S_j$ are at least $\sqrt{n}/2$ positions apart. □

**Lemma 11.** *Any two occurrences of suffixes $S_{j_1} \neq S_{j_2}$ have distinct right endpoints.*

*Proof.* Assume the opposite. Let $|S_{j_1}| > |S_{j_2}|$. By the definition, $S_{j_1}$ is the shortest suffix of $T[1..j_1]$ such that its minimal period at least $\sqrt{n}/2$. As $S_{j_1}$ and $S_{j_2}$ have occurrences with equal right endpoints, $S_{j_2}$ is a suffix of $S_{j_1}$, and, as a corollary, of $T[1..j_1]$. But, $S_{j_2}$ is shorter than $S_{j_1}$ and its minimal period is larger than $\sqrt{n}/2$. A contradiction. □

The algorithm will make use of sorted lists of occurrences of distinct suffixes $S_j$. From above it follows that the length of one list is at most $2\sqrt{n}$, whereas the total length of the lists is at most $n$. We compute the lists in the following way. Suppose that each $S_j$ is replaced with an integer $u_j \in [1, 2n]$ so that $S_{j_1} \neq S_{j_2}$ implies $u_{j_1} \neq u_{j_2}$. We create an array of $2n$ empty lists and scan positions of $T$ from the left to the right. For each $j = 1..n$ we add position $j$ to the list $u_j$. Thus, we can compute the lists in $\mathcal{O}(n)$ time. We now describe the replacement procedure.

**Lemma 12.** *Given $j$, the length of $S_j$ can be computed in $\mathcal{O}(\sqrt{n})$ time.*

*Proof.* We start by computing the minimal periods of suffixes $T[j - \sqrt{n} + 1..j], T[j - \sqrt{n} + 2..j], \ldots, T[j]$. This can be done by constructing the border array $B$ of the reverse of $T[j - \sqrt{n} + 1..j]$ in $\mathcal{O}(\sqrt{n})$ time: The minimal period of $T[j - \ell..j]$ will be equal to $\ell - B[\ell]$. If the minimal period $\pi$ of $T[j - \sqrt{n} + 1..j]$ is at least $\sqrt{n}/2$, then $S_j$ is one of the suffixes and we already know it. Suppose that $\pi \leq \sqrt{n}/2$. Let $T[k..j]$ be the longest suffix of $T[1..j]$ such that its minimal period equals $\pi$. We can compute $T[k..j]$ in constant time (Query 4 of Lemma 2). We claim that the minimal period $\gamma$ of $T[k - 1..j]$ is larger than $\sqrt{n}/2$. Indeed, $T[k..j]$, as a suffix of $T[k-1..j]$, is periodic with period $\gamma$. If $\gamma \leq \sqrt{n}/2$, we have $\pi + \gamma \leq \sqrt{n} \leq T[k..j]$. By the Periodicity lemma, $T[k..j]$ is periodic with period $\gcd(\pi, \gamma)$. Because of the minimality of $\pi$, $\gamma$ must be a multiple of $\pi$. It follows that $T[k - 1..j]$ has period $\pi$, which contradicts the definition of $T[k..j]$. □

For each $j = 1..n$ we compute the length of $S_j$. We then build the suffix tree of the reverse of $T$. The suffix tree contains at most $2n$ nodes which we enumerate from 1 to $2n$. As we know, each position can be the right endpoint of an occurrence of at most one $S_j$. This suggests the following algorithm. We consider the leaves of the tree from left to the right. Let the current leaf be labeled by $T[j]T[j-1]\dots T[1]$. We follow the path from the leaf to the root to find the highest branching node of string depth $\geq |S_j|$. Leaves in the subtree of this node will correspond to the positions $j'$ such that $S_{j'} = S_j$. We replace all $S_{j'}$ in the subtree with the order number of the node and proceed to the leftmost suffix outside the subtree. The replacement procedure takes $\mathcal{O}(n)$ time overall.

*The check.* Throughout stage $k$ we maintain, for all $j \in J_k$, a pointer to the last position in the list of $u_j$ that has been explored by the algorithm. A long border (if any) of the candidate substring $T[i..j]$ must be induced by an occurrence of $S_j$ in the interval $[i, j]$. The algorithm explores occurrences in the list of $S_j$ in turn starting from the one it stopped at. For each occurrence $p \geq i$ the algorithm compares substrings $F_1 = T[i..p+|S_j|-1]$ and $F_2 = T[j-|F_1|+1..j]$ (Query 1 of Lemma 2). If they are equal, $T[i..j]$ has a long border. Otherwise, the algorithm proceeds to the next occurrence in the list. If no occurrence in the list induces a long border, $T[i..j]$ is unbordered.

### 4.3 Pseudocode and the bounds

To summarize, we give pseudocode of stage $k$ of algorithm $\mathcal{B}$. Preprocessing for long border checks (computation of suffixes $S_j$ and their lists) is done before the first stage (not shown).

**Algorithm 1** Stage $k$ of Algorithm $\mathcal{B}$.

```
1: Build the suffix tree of $T$ and $T_k = T[(k-1)\sqrt{n}+1..(k+1)\sqrt{n}]$
2: for $i = 1..n$ do
3:     Compute the longest prefix $P_i$ of $T[i..n]$ that occurs in $T_k$
4: for $\ell = (k-1)\sqrt{n}+1..(k+1)\sqrt{n}$ do
5:     Compute the shortest border array of $T[\ell..(k+1)\sqrt{n}]$ \$ $T[(k-1)\sqrt{n}+1..\ell-1]$
6:     Sort the array
7:     Compute the values $r_\ell^p$
8: for $i = 1..(k-3)\sqrt{n}$ do
9:     $\ell \leftarrow$ position of an occurrence of $P_i$ in $T_k$
10:    $j \leftarrow r_\ell^{|P_i|}$
11:    if $T[i..j]$ does not have a long border then
12:        Update LongestUnbordered
```

**Theorem 2.** *The* worst-case *time complexity of algorithm $\mathcal{B}$ is $\mathcal{O}(n^{1.5})$. The space complexity of the algorithm is $\mathcal{O}(n)$.*

*Proof.* It suffices to show that one stage of the algorithm takes $\mathcal{O}(n)$ time. Preprocessing takes $\mathcal{O}(n)$ time. For each position $i = 1..n$ we spend constant time plus the time needed for the long border check. The total amount of time needed for the long border checks is linear in total length of the lists, which is at most $n$, as we never check an occurrence in a list twice for any position of $T$. □

## References

1. R. Assous and M. Pouzet. Une caractérisation des mots périodiques. *Journal of Discrete Mathematics*, 25(1):1–5, 1979.
2. M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer-Verlag, 2000.
3. M.A. Bender and M. Farach-Colton. The level ancestor problem simplified. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, pages 508–515. Springer-Verlag, 2002.
4. J.-P. Duval. Relationship between the period of a finite word and the length of its unbordered segments. *Journal of Discrete Mathematics*, 40(1):31–44, 1982.
5. J.-P. Duval, T. Lecroq, and A. Lefebvre. Linear computation of unbordered conjugate on unordered alphabet. *Journal of Theoretical Computer Science*, 522(0):77–84, 2014.
6. A. Ehrenfeucht and D.M. Silberger. Periodicity and unbordered segments of words. *Journal of Discrete Mathematics*, 26(2):101–109, 1979.
7. M. Farach-Colton. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE Computer Society, 1997.
8. Š. Holub and D. Nowotka. The Ehrenfeucht–Silberger problem. *Journal of Combinatorial Theory, Series A*, 119(3):668–682, 2012.
9. L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, 2010.
10. G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Oct 1989.
11. G. Kucherov, A. Loptev, and T. Starikovskaya. On Maximal Unbordered Factors. *ArXiv e-prints*, 2015. To appear in CPM'2015.
12. J.H. Morris Jr. and V.R. Pratt. A linear pattern-matching algorithm, report 40. Technical report, University of California, Berkeley, 1970.
13. W. Szpankowski. *Average Case Analysis of Algorithms on Sequences.* John Wiley & Sons, Inc., 2001.
14. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science*, pages 1–11, 1973.