Efficient Data Structures for the Factor Periodicity Problem

Tomasz Kociumaka[1], Jakub Radoszewski[1,⋆],
Wojciech Rytter[1,2,⋆⋆], and Tomasz Waleń[3,1]

[1] Faculty of Mathematics, Informatics and Mechanics,
University of Warsaw, Warsaw, Poland
{kociumaka,jrad,rytter,walen}@mimuw.edu.pl
[2] Faculty of Mathematics and Computer Science,
Copernicus University, Toruń, Poland
[3] Laboratory of Bioinformatics and Protein Engineering,
International Institute of Molecular and Cell Biology in Warsaw, Poland

Abstract. We present several efficient data structures for answering queries related to periods in words. For a given word w of length n the Period Query given a factor of w (represented by an interval) returns its shortest period and a compact representation of all periods. Several algorithmic solutions are proposed that balance the data structure space (ranging from $O(n)$ to $O(n \log n)$), and the query time complexity (ranging from $O(\log^{1+\varepsilon} n)$ to $O(\log n)$).

1 Introduction

Computation of different types of periodicities is one of the central parts of algorithmics on words. In this paper we consider periods of factors of words. More precisely, we show a data structure that allows to find the smallest period and a compact representation of all periods of a factor given by an interval of positions. By a compact representation we mean a logarithmic number of integers representing a small set of arithmetic progressions.

A similar type of queries (for tiling periodicity) was studied in [7]. Also a few results for primitivity queries were known (testing if a factor is primitive): $O(\log n)$ time for queries with $O(n \log^{\varepsilon} n)$ space, see [2], and $O(1)$-time queries with $O(n \log n)$ space, see [7].

We consider words over an integer alphabet Σ. For a word $w = a_1 a_2 \dots a_n$ denote by $w[l, r]$ the factor $a_l a_{l+1} \dots a_r$. We say that an integer p is a *period* of w if $a_i = a_{i+p}$ holds for all $1 \le i \le n - p$. Denote by $\mathit{MinPer}(l, r) = \mathsf{per}(w[l, r])$ the smallest period of the word $w[l, r]$, and by $\mathit{AllPer}(l, r)$ denote the set of all periods of $w[l, r]$. It is a known fact that the set $\mathit{AllPer}(l, r)$ can be represented as a union of a logarithmic number of pairwise disjoint sets, each set forming an arithmetic progression (a proof of this fact can also be inferred from our paper), see Fig. 1. We present a series of algorithms for the following problem.

[⋆] The author is supported by grant no. N206 568540 of the National Science Centre.
[⋆⋆] The author is supported by grant no. N206 566740 of the National Science Centre.

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 284–294, 2012.

Input: Store a word w of size n;

Queries: Given $1 \le l < r \le n$ compute:

$\mathit{MinPer}(l, r)$ – the smallest period of $w[l, r]$ and
$\mathit{AllPer}(l, r)$ – a logarithmic size representation of all periods of $w[l, r]$.

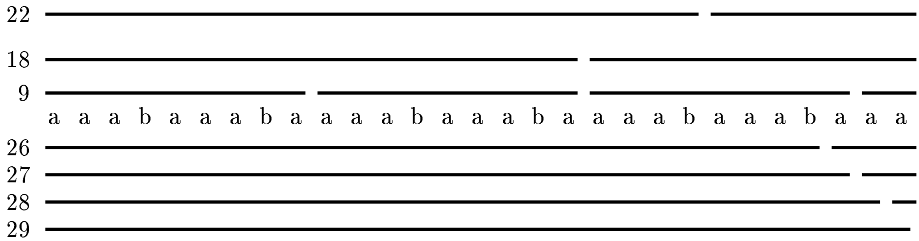

Fig. 1. A word $w = w[1, 29]$ together with its periods. We have $\mathit{MinPer}(1, 29) = 9$ and $\mathit{AllPer}(1, 29)$ can be decomposed into a union of three arithmetic progressions: $\{9, 18\} \cup \{22\} \cup \{26, 27, 28, 29\}$. We also have $\mathit{AllPer}(1, 3) = \{1, 2, 3\}$ and $\mathit{AllPer}(1, 7) = \{4, 5, 6, 7\}$.

Our results are presented in the following table, we obtain a kind of trade-off between data structure space and query time.

Data structure space	Query time
$O(n)$	$O(\log^{1+\varepsilon} n)$
$O(n \log\log n)$	$O(\log n (\log\log n)^2)$
$O(n \log^{\varepsilon} n)$	$O(\log n \log\log n)$
$O(n \log n)$	$O(\log n)$

In different algorithms we use the classical textual data structures: the Dictionary of Basic Factors (DBF) and the suffix tree [1,3,6].

Given a word w of length n, the *basic factors* of w are its factors of lengths which are powers of two. The DBF assigns integer identifiers from the range $[1, n]$ to all basic factors, so that different basic factors of the same length receive different identifiers. The DBF uses $O(n \log n)$ time and space to construct.

The suffix tree of w, denoted here as $T(w)$, is a compacted trie representing all factors of w. Each factor of w corresponds to an explicit or implicit node of $T(w)$. For any explicit node v of $T(w)$, by $\mathit{val}(v)$ we denote the factor of w corresponding to this node. If w is extended with an end-marker then each leaf of $T(w)$ corresponds to a suffix of w, hence we can store an array $\mathit{leaf}[i]$ that assigns, to each suffix $w[i, n]$, the leaf it corresponds to. Recall that $T(w)$ has size $O(n)$ and can be constructed in $O(n)$ time [1,3,4,6].

2 Combinatorics of Periods, Borders and Prefix-Suffixes

The word u is a *border* of the word w if u is both a prefix and a suffix of w. The following well-known observation connects the notions of a border and a period, see [1,3].

Observation 1. *The word w has a period p if and only if w has a border of length $|w| - p$.*

Due to this observation, in the Period Queries we will actually compute *MaxBorder*(l, r), the length of the longest border of $w[l, r]$, and *AllBorders*(l, r), a representation of the set of lengths of all borders of $w[l, r]$ as a union of a logarithmic number of arithmetic progressions, instead of *MinPer*(l, r) and *AllPer*(l, r) respectively.

If there is no ambiguity we sometimes write u, *AllBorders*(u), instead of $w[l, r]$, and *AllBorders*(l, r), where $u = w[l, r]$.

Throughout the paper we use the following classical fact related to periods.

Fact 1 (Periodicity lemma [1,3,5]). *If a word of length n has two periods p and q, such that $p + q \leq n + \gcd(p, q)$, then $\gcd(p, q)$ is also a period of the word.*

Denote by *BordersLarger*(u, M) the set of elements of *AllBorders*(u) larger than M. The periodicity lemma easily implies the following fact.

Lemma 1. *If $M \geq |u|/2$ then BordersLarger(u, M) is a single arithmetic progression.*

Proof. Any border of u from the set *BordersLarger*(u, M) corresponds, by Observation 1, to a period of u smaller than $|u|/2$. By the periodicity lemma, all such periods are multiples of $\mathsf{per}(u)$, hence they form an arithmetic progression, hence the elements of *BordersLarger*(u, M) form a single arithmetic progression. □

Denote by $\mathcal{BF}(w)$ the set of basic factors of w, recall that these are the factors of lengths which are powers of two.

A *prefix-suffix* of a pair of words (x, y) is a word z which is a prefix of x and a suffix of y (see Fig. 2). If $x = y$ then the notion of a *prefix-suffix* corresponds to that of the border of a word. Assume that $|x| = |y|$. A prefix-suffix z of (x, y) is *large* if $|z| > \frac{1}{2}|x|$. For any two factors $x, y \in \mathcal{BF}(w)$ of the same length denote by *LargePS*(x, y) the set of all lengths of large prefix-suffixes of (x, y).

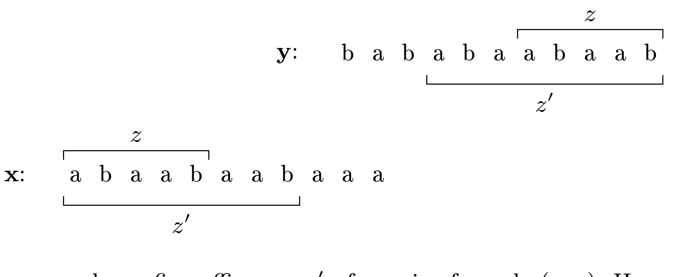

Fig. 2. Two example prefix-suffixes z, z' of a pair of words (x, y). Here only z' is a large prefix-suffix.

Lemma 2. *Assume $x, y \in \mathcal{BF}(w)$, $|x| = |y|$. Then $LargePS(x, y)$ forms a single arithmetic progression.*

Proof. Let $M = \max LargePS(x, y)$. Let u be the suffix of y of length M. Then $LargePS(x, y) = BordersLarger(u, |x|/2)$. The conclusion follows directly from Lemma 1. □

3 Main Algorithm

We show how Period Queries can be reduced to simpler queries that we introduced in the previous section: *LargePS* and *BordersLarger* queries. In the following sections we discuss data structures for answering these queries.

Denote $Max2Power(k) = 2^i$, where 2^i is the largest power of two not exceeding k. For a set of integers X and an integer k denote

$$k \ominus X \ = \ \{k - x \ : \ x \in X\}, \qquad k \oplus X \ = \ \{k + x \ : \ x \in X\}.$$

We break the Period Queries into a series of smaller queries of the form $LargePS(x_i, y_i)$ related to basic factors x_i, y_i.

```
Algorithm MAIN($(l, r)$)  {computes $AllPer(l, r)$}

  – $Borders := \emptyset$; $u := w[l, r]$
  – for each $(x_i, y_i) \in \mathcal{I}(u)$ do
          $Borders := Borders \cup LargePS(x_i, y_i)$
  – $Borders := Borders \cup BordersLarger(u, Max2Power(|u|))$
  – return $|u| \ominus Borders$
```

In the algorithm $\mathcal{I}(u)$ denotes a set of pairs of a prefix and a suffix of u of lengths which are increasing powers of two (i.e. prefixes and suffixes of u which are basic factors), see Fig. 3.

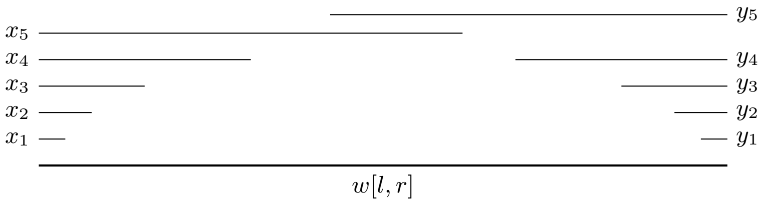

Fig. 3. $\mathcal{I}(w[l, r])$. For each i we have $x_i, y_i \in \mathcal{BF}(w)$.

Lemma 3. *A Period Query for $AllPer(l, r)$ and $MinPer(l, r)$ can be answered using a logarithmic number of queries of the type $LargePS(x, y)$ for $x, y \in \mathcal{BF}(w)$, $|x| = |y|$, and single query of the type $BordersLarger(u, M)$ for $M = Max2Power(|u|)$.*

4 Implementation of *LargePS* and *BordersLarger*

For a given word w, its factor $v = w[l,r] \in \mathcal{BF}(w)$ (given by pair (l,r)) and number i, we introduce the following type of query:

> $SUCC(i,v)$ ($PRED(i,v)$): find the minimal (maximal) index j in range $[i, i+|v|]$ ($[i-|v|, i]$) such that $w[j, j+|v|-1] = v$

We will show how to implement *LargePS* and *BordersLarger* using a small number of $SUCC$ and $PRED$ queries. First we introduce one more combinatorial tool.

Denote by $Occ(v,w)$ the set of starting positions of all occurrences of the word v within the word w. The following fact is a folklore consequence of the periodicity lemma.

Fact 2. *Consider two non-empty words x, y such that $|y| \le 2\cdot|x|$. Then $Occ(x,y)$ forms a single arithmetic progression. If, moreover, $|Occ(x,y)| \ge 3$, the difference of this progression equals $\mathsf{per}(x)$.*

A straightforward application of this fact is the computation of the representations of the *Occ* sets using $SUCC$ and $PRED$ queries.

Lemma 4. *If x, y are factors of w such that $|y| \le 2\cdot|x|$ then a constant-size representation of the set $Occ(x,y)$ (as an arithmetic progression) can be computed using $O(1)$ $SUCC/PRED$ queries in w.*

Proof. Let $y = w[i,j]$. First we perform two $SUCC$ queries: $SUCC(i,x) = p$ and $SUCC(p,x) = q$. If p or q does not exist, we are done. By Fact 2, $Occ(x,y)$ is an arithmetic progression. From p and q we obtain the first element and the difference of this progression. Finally, we use a $PRED(j-|x|+1, x)$ query to find the last element of the progression. □

We proceed with the implementation of *LargePS*(x,y). For this we provide a more detailed characterization of this set which turns out to be crucial for the construction of an efficient algorithm.

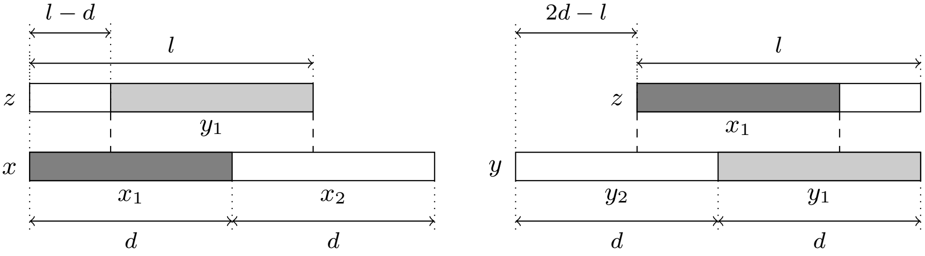

Fig. 4. A pair (x,y) has a large prefix-suffix z of length l if and only if y_1 and x_1 occur at certain positions in x and y, respectively

Lemma 5. *Let $x, y \in \mathcal{BF}(w)$ and $|x| = |y| > 1$. Also let $x = x_1x_2$ and $y = y_2y_1$, where $|x_1| = |x_2| = |y_1| = |y_2| = d$. Then:*

$$\mathit{LargePS}(x, y) = ((2d + 1) \ominus \mathit{Occ}(x_1, y)) \cap ((d - 1) \oplus \mathit{Occ}(y_1, x)) \setminus \{d\}. \quad (1)$$

Proof. Note that if $l > d$, then $l \in \mathit{LargePS}(x, y)$ if and only if $l - d + 1 \in \mathit{Occ}(y_1, x)$ and $2d - l + 1 \in \mathit{Occ}(x_1, y)$, see Fig. 4. □

Below we show one additional property, see Fig. 5, of the sets considered in Lemma 5 that we use in an algorithm computing $\mathit{LargePS}(x, y)$. This property is used for constant-time computation of intersection of related arithmetic progressions.

Lemma 6. *Assume $x, y \in \mathcal{BF}(w)$, $|x| = |y| > 1$ and $x = x_1x_2$ and $y = y_2y_1$, $|x_1| = |x_2| = |y_1| = |y_2|$. If $|\mathit{Occ}(x_1, y)| \geq 3$ and $|\mathit{Occ}(y_1, x)| \geq 3$ then $\mathsf{per}(x_1) = \mathsf{per}(y_1)$, that is, the arithmetic progressions $\mathit{Occ}(x_1, y)$ and $\mathit{Occ}(y_1, x)$ have the same difference.*

Proof. Let $p = \mathsf{per}(x_1)$ and $p' = \mathsf{per}(y_1)$. Assume to the contrary that $p > p'$. Let $l = \max \mathit{Occ}(x_1, y)$. The size of $\mathit{Occ}(x_1, y)$ implies that the length of the overlap of this occurrence of x_1 and y_1 is at least $2p$. This overlap corresponds to a suffix of x_1 having periods p and p', hence, by the periodicity lemma, having period $d = \gcd(p, p')$. This concludes that x_1 has period $d < p$, a contradiction. The case of $p < p'$ can be treated similarly, by considering the leftmost occurrence of y_1 within x. □

Observation 2. *Assume we have compact representations (as integer triples: the first and the last element and the difference) of two arithmetic progressions with the same difference (up to absolute value). Then we can compute a compact representation of their intersection in constant time.*

```
Algorithm Compute $\mathit{LargePS}(x, y)$   $\{x, y \in \mathcal{BF}(w),\ |x| = |y|\}$

 – if $|x| = 1$ then just check if $x = y$
 – let $x = x_1x_2$, $y = y_2y_1$ be such that $|x_1| = |x_2| = |y_1| = |y_2| = d$
 – compute (using Lemma 4)
        $\mathcal{S}_1 = (2d + 1) \ominus \mathit{Occ}(x_1, y)$  and  $\mathcal{S}_2 = (d - 1) \oplus \mathit{Occ}(y_1, x)$
 – if $|\mathcal{S}_1| \leq 2$ or $|\mathcal{S}_2| \leq 2$ then (non-periodic case)
     compute $\mathcal{S}_1 \cap \mathcal{S}_2$ by checking all elements of the smaller set
 – else (periodic case, apply Lemma 6)
     compute $\mathcal{S}_1 \cap \mathcal{S}_2$ in $O(1)$ time as an intersection of two
        arithmetic progressions with the same difference
 – return $(\mathcal{S}_1 \cap \mathcal{S}_2) \setminus \{d\}$
```

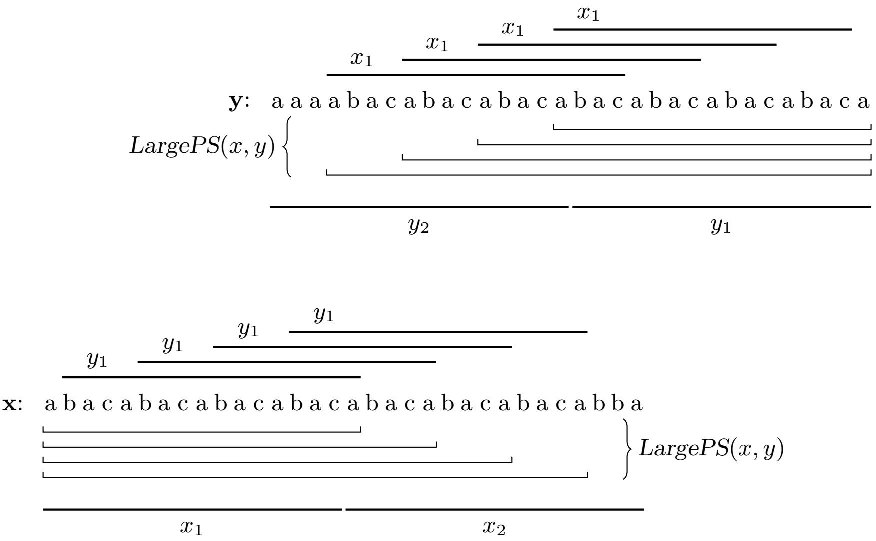
x_1
x_1
x_1
x_1
y: a a a a b a c a b a c a b a c a b a c a b a c a b a c a b a c a
LargePS(x, y)
y_2
y_1
y_1
y_1
y_1
y_1
x: a b a c a b a c a b a c a b a c a b a c a b a c a b a c a b b a
LargePS(x, y)
x_1
x_2

Fig. 5. *LargePS*$(x, y) = \{17, 21, 25, 29\}$ is an arithmetic progression determined by *Occ*$(x_1, y) = \{4, 8, 12, 16\}$ and *Occ*$(y_1, x) = \{2, 6, 10, 14\}$. Both progressions have the same difference.

The *SUCC*/*PRED* queries are used only to compute compact representations of the *Occ* sets. We conclude with the following lemma.

Lemma 7. *Assume* $x, y \in \mathcal{BF}(w)$, $|x| = |y|$. *Then LargePS*(x, y) *can be computed using* $O(1)$ *SUCC*/*PRED queries and* $O(1)$ *additional operations.*

Finally we show how to implement *BordersLarger*(u, M) queries required in the MAIN algorithm.

Lemma 8. *For each factor* u *of word* w *and* $M =$ *Max2Power*$(|u|)$*, the set BordersLarger*(u, M) *can be computed (as a single arithmetic progression) by a constant number of SUCC*/*PRED queries.*

Proof. By the proof of Lemma 1, all the elements of the set *BordersLarger*(u, M) correspond to multiples of the smallest period of u and that this set can be non-empty only if $\mathsf{per}(u) < |u| - M$, which is not greater than $\frac{1}{2}|u|$.

Let x be the prefix of u of length M. Its first occurrence in u is an occurrence as a prefix. Using *SUCC* query we locate the second occurrence. If there is none, the result is empty. Otherwise, let d be the difference between the starting positions of these occurrences.

Then d is the only candidate for the smallest period of u smaller than $|u| - M$, if there is any. Indeed, if $p = \mathsf{per}(u) < d$ then x would occur earlier, at the position p. If $d < p \leq |u| - M$ then the prefix of u of length $d + M$ would have the periods d and p, hence, by the periodicity lemma, the period $d' = \gcd(d, p)$, which concludes that $d' < p$ would be a period of u, which is not possible.

We need to check if d is a period of u, we know that it is a period of x. It suffices to check a similar condition to the previous one, but from the end of u and using a *PRED* query. Let y be a suffix of u of length M. With a *PRED* query we find the previous occurrence of y as a factor of u. If this occurrence exists and the difference between these occurrences equals d, then d is a period of y and, since x and y cover u, d is a period of u. Otherwise d cannot be a period of u.

In conclusion, we either obtain an empty set $\mathit{BordersLarger}(u, M)$ or a progression with difference d. □

Now it suffices to show how to implement the $\mathit{SUCC}/\mathit{PRED}$ queries efficiently. Two ways to do this are described in the following section. Here we set up some intuition by giving an $O(n \log n)$ space and $O(\log n)$ query time solution.

We will use the Dictionary of Basic Factors. For each basic factor we store an array of its occurrences in ascending order. These arrays are accessed by factors' length and DBF identifier, e.g. $A[k][\mathit{id}(v)]$ is an array for a factor v of length 2^k with identifier $\mathit{id}(v)$. Clearly, the total size of these arrays is $O(n \log n)$ and they can be constructed in $O(n \log n)$ time from the DBF. To compute $\mathit{SUCC}(i, v)$, we perform a binary search in the array corresponding to v in order to find the first occurrence of v that is not less than i. The *PRED* queries are answered analogously. Hence, we obtain $O(\log n)$ query time.

As a conclusion of Lemmas 3, 7 and 8, we get the following result. It is improved in the next section.

Theorem 1. *A word w of length n can be stored in an $O(n \log n)$ space data structure so that the Period Queries can be answered in $O(\log^2 n)$ time. This data structure can be constructed in $O(n \log n)$ time.*

5 Implementation of *PRED/SUCC* Queries

In this section we present various implementations of the queries *PRED* and *SUCC*. The query time decreases at the cost of an increase in space complexity.

5.1 Improving Query Time Using DBF

Here we show an $O(n \log n)$ space data structure with $O(1)$ query time for $\mathit{PRED}/\mathit{SUCC}$. It improves the very simple solution described in the end of the previous section. The data structure remains simple and also uses the Dictionary of Basic Factors. Combined with Lemmas 3, 7 and 8, this yields an $O(n \log n)$ space data structure for answering Period Queries in $O(\log n)$ time.

Lemma 9. *A word w of length n can be stored in an $O(n \log n)$ space data structure, so that the queries $\mathit{SUCC}(i, v)$ and $\mathit{PRED}(i, v)$ for $v \in \mathcal{BF}(w)$ can be answered in $O(1)$ time. Moreover, this data structure can be constructed in $O(n \log n)$ expected time.*

Proof. We start by computing DBF identifiers $\mathit{id}(v)$ for all $v \in \mathcal{BF}(w)$. The set of occurrences of each v is then divided into $\left\lceil \frac{n}{|v|} \right\rceil$ sets $\mathit{Occ}_{v,0}$, $\mathit{Occ}_{v,1}$, … (some of them possibly empty). The $\mathit{Occ}_{v,j}$ set stores the occurrences of v starting in the range $[j \cdot |v|, (j+1) \cdot |v|)$. By Fact 2, each set $\mathit{Occ}_{v,j}$ is either empty or can be represented as an arithmetic progression.

We prepare a perfect hash table $\mathcal{H}$: for each triple $(|v|, \mathit{id}(v), j)$ such that $\mathit{Occ}_{v,j} \neq \emptyset$ we store an $O(1)$ space representation of the arithmetic progression formed by $\mathit{Occ}_{v,j}$. The total number of occurrences of factors $v \in \mathcal{BF}(w)$ in the word w is $O(n \log n)$, therefore $\mathcal{H}$ takes $O(n \log n)$ space and can be constructed in $O(n \log n)$ expected time.

The $\mathit{SUCC}(i, v)$ queries can be answered in $O(1)$ time by inspecting a constant number of entries of the hash table. Observe that the range $[i, i + |v|]$ is covered by exactly 2 intervals of the form $[j \cdot |v|, (j+1) \cdot |v|)$. Therefore we find and return the successor of i among the elements of the corresponding arithmetic progressions $\mathit{Occ}_{v,j}$, $\mathit{Occ}_{v,j+1}$. The *PRED* queries are answered similarly. □

We obtain the aforementioned result.

Theorem 2. *A word w of length n can be stored in an $O(n \log n)$ space data structure so that the Period Queries can be answered in $O(\log n)$ time. This data structure can be constructed in $O(n \log n)$ expected time.*

5.2 Space Reductions Using Range Predecessor Queries

In this section we present another approach to *PRED*/*SUCC* queries. It gives slightly worse query time, but the space usage is significantly better. This method is based on the results of [9] and [8] instead of the DBF.

Recall that $T(w)$ is the suffix tree of w. Our main tool is the following data structure described in a recent paper by Nekrich and Navarro [9].

Lemma 10. [Range Predecessor/Successor Queries, page 9 in [9]] *A word w of length n can be stored in an $O(f(n))$ space data structure so that for a node v of $T(w)$ and position j within w, the values $\mathit{PRED}(j, \mathrm{val}(v))$ and $\mathit{SUCC}(j, \mathrm{val}(v))$ can be computed in $O(g(n))$ time for:*

- *$f(n) = O(n)$ and $g(n) = O(\log^{\varepsilon} n)$*
- *$f(n) = O(n \log\log n)$ and $g(n) = O((\log\log n)^2)$*
- *$f(n) = O(n \log^{\varepsilon} n)$ and $g(n) = O(\log\log n)$.*

The other tool is the following data structure for weighted trees proposed by Kopelowitz and Lewenstein [8].

Lemma 11. [Weighted Level Ancestor Queries [8]] *Let T be a tree of n nodes with positive integer weights up to $O(n)$ in edges. We can store T in an $O(n)$ space data structure that can answer the following queries in $O(\log\log n)$ time: Given an integer h and a node v such that the distance from the root to v is greater than h, return the highest ancestor of v whose distance to the root is at least h.*

As a corollary, we obtain the following theorem.

Theorem 3. *A word w of length n can be stored in an $O(f(n))$ space data structure, so that the Period Queries can be answered in $O(g(n))$ time for:*

- $f(n) = O(n)$ *and* $g(n) = O(\log^{1+\varepsilon} n)$
- $f(n) = O(n \log\log n)$ *and* $g(n) = O(\log n (\log\log n)^2)$
- $f(n) = O(n \log^{\varepsilon} n)$ *and* $g(n) = O(\log n \log\log n)$*.*

Proof. Our goal is to obtain the space and query time of $SUCC/PRED$ queries as in Lemma 10. Then we can complete the proof by using, as previously, Lemmas 3, 7 and 8.

For a factor $v = w[l,r]$ let $locus(v)$ be a node of $T(w)$ such that

$$Occ(v,w) = Occ(val(locus(v)), w).$$

If we know $locus(v)$ then the $PRED/SUCC$ queries for v can be replaced by $PRED/SUCC$ queries for $locus(v)$:

$$PRED(j,v) = PRED(j, val(locus(v)))$$
$$SUCC(j,v) = SUCC(j, val(locus(v))).$$

By Lemma 10, such queries can be answered efficiently. Therefore it suffices to show how to find efficiently $locus(v)$, given the interval $[l,r]$ such that $v = w[l,r]$.

Let us introduce edge lengths in $T(w)$ as distances in the underlying trie, i.e. the lengths of factors of w that have been compactified to the corresponding edges. Recall that $leaf[l]$ points to the leaf that corresponds to the suffix $w[l,n]$. Note that $locus(v)$ is the highest ancestor of $leaf[l]$ whose distance to the root is at least $r-l+1$. Finding such ancestor of this leaf can be described in terms of Weighted Level Ancestor Queries, and we can apply Lemma 11. Note that both the space and the query time of this data structure is dominated by the Range Predecessor/Successor Queries. This completes the proof. □

6 Final Remarks

The algorithm that we presented spends most of the time computing very short borders, that correspond to very large periods. If we are interested in periods of $u = w[l,r]$ which are smaller than $(1-\delta)|u|$ for some $\delta > 0$, then we need to consider only a constant number of elements from $\mathcal{I}(u)$. Hence, the queries are faster by a multiplicative $O(\log n)$ factor. In particular, for a data structure of $O(n)$ space the queries work in $O(\log^{\varepsilon} n)$ time and for a data structure of $O(n \log n)$ space the query time is $O(1)$.

Note that this is the case in the problem of primitivity testing, in which we are to check if a factor $w[l,r]$ has a non-trivial period that divides the length of the factor. Here $\delta = \frac{1}{2}$. We conclude with the following corollary.

Corollary 1. *A word w of length n can be stored in an $O(n)$ space data structure so that the primitivity queries can be answered in $O(\log^{\varepsilon} n)$ time, or in an $O(n \log n)$ space data structure with $O(1)$ query time.*

References

1. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
2. Crochemore, M., Iliopoulos, C., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting Powers and Periods in a String from Its Runs Structure. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 258–269. Springer, Heidelberg (2010)
3. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2003)
4. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS, pp. 137–143. IEEE Computer Society (1997)
5. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society 16, 109–114 (1965)
6. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
7. Karhumäki, J., Lifshits, Y., Rytter, W.: Tiling periodicity. Discrete Mathematics & Theoretical Computer Science 12(2), 237–248 (2010)
8. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: Bansal, N., Pruhs, K., Stein, C. (eds.) SODA, pp. 565–574. SIAM (2007)
9. Nekrich, Y., Navarro, G.: Sorted Range Reporting. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 271–282. Springer, Heidelberg (2012)