An Algorithmic Toolbox for Periodic Partial Words[☆]

Florin Manea[a,*], Robert Mercaş[a,*], Cătălin Tiseanu[b]

[a]*Christian-Albrechts-Universität zu Kiel, Institut für Informatik, Christian-Albrechts-Platz 4, D-24098 Kiel, Germany*
[b]*Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, RO-010014 Bucharest, Romania*

Abstract

This work presents efficient solutions to several basic algorithmic problems regarding periodicity of partial words. In the first part of the paper, we show that all periods of a partial word of length n are determined in $\mathcal{O}(n \log n)$ time. Moreover, we define algorithms and data structures that help us answer in constant time queries regarding the periodicity of a word's factors. For these we need an $\mathcal{O}(n^2)$ preprocessing time and an $\mathcal{O}(n)$ updating time, whenever the words are extended by adding a letter. In the second part of the paper, we show that identifying a way to construct a periodic partial word by substituting the letters on some positions of a full word w with holes, where the distance between two consecutive holes must be greater than a given number, can be done in optimal time $\mathcal{O}(|w|)$. We also show that identifying a substitution which replaces the minimum number of positions by holes can be done as fast as in the previous case.

Keywords: Combinatorics on Words, Periodicity, Partial Words, Algorithms

[☆]This work represents an extended version of a paper presented at the *36th International Symposium on Mathematical Foundations of Computer Science, MFCS 2011*, [1].

[*]Corresponding authors.

Email addresses: `flm@informatik.uni-kiel.de` (Florin Manea), `rgm@informatik.uni-kiel.de` (Robert Mercaş), `ctiseanu@gmail.com` (Cătălin Tiseanu)

1. Introduction

Periodicity is one of the most fundamental properties of words. Problems correlated to periodicity computation have applications in formal languages and automata theory, algorithmic combinatorics on words, data compression, string searching and pattern matching algorithms (see [2, 3, 4] and the references therein). The first idea of a fast algorithm identifying all periods of a word was given in [5]. Almost a decade later, in [6], Crochemore provides the first correct time-space optimal algorithm taking into account also the fact that the alphabet is ordered. This solution, as many other subsequent efficient solutions of the problem, relies heavily on the possibility of solving in linear time and space the string matching problem: that is, finding all the occurrences of a shorter word (called pattern) in a larger one (called text) in linear time and space. Solutions that fulfil these efficiency requirements were given, for instance, in [7, 8].

For partial words, sequences that beside regular symbols contain some "holes" or "don't care symbols" that can be substituted with any letter of the word's alphabet, the concept of periodicity was also deeply analysed ([2] surveys most of the work in this area and discusses the obtained results in comparison with the ones existing for words). To start with, the problem of testing the primitivity of a partial word (i.e., checking whether a given partial word has no period that divides its length) was discussed in [9, 10], where partial solutions were proposed; similarly to the classical case, these solutions were based on (partial) words matching algorithms. To this end, we recall that fast partial words matching algorithms were provided in [11], starting from the ideas initiated in [12]. The fastest deterministic partial matching algorithms, known so far, solve this problem in $\mathcal{O}(n \log n)$ time (see also [13]).

The study of repetitions in partial words was developed in [14]. A string is said to contain a repetition if it has consecutive factors compatible with the same full word. In [14], it is proved that over a binary alphabet there exist infinite partial words that are cube-free, which, in other words, means that for all factors the periods are greater than one third of their length. In [15] the authors solve a conjecture regarding the minimum size alphabet needed in order to construct an infinite partial word that remains overlap-free even after arbitrarily many insertions of holes. That is to say, here, all factors of the infinite word have periods greater than half their length. The authors use, in order to prove this, an $\mathcal{O}(nd)$ algorithm that determines if, after hole insertion such that between each two holes there are at least d non-hole symbols, a word has a certain period.

Algorithms regarding freeness of factors of partial words, factors free of some property, were firstly discussed in [14, 16]. In [16] the authors construct some data structures which enable them to answer, after a preprocessing phase done in $\mathcal{O}(n^2)$, queries regarding the freeness of their factors in constant time. Moreover, the authors provide a method for updating the data structures in $\mathcal{O}(n \log n)$ time, whenever a symbol is concatenated to the right end of the existing string, and still being able to answer the queries in constant time.

This paper proposes a series of algorithms, for some of the most basic problems related to periodicity in partial words, more efficient than the already existing ones, and discusses possible generalisations of these problems. After presenting some basic concepts regarding partial words and periodicity, in the next section, our paper continues with two main parts.

First, in Section 3 results from [9, 10, 16] are extended and improved. The main result of that section is an algorithm that computes all periods of a partial word of length n in $\mathcal{O}(n \log n)$ time. Further, motivated by natural questions like finding efficiently all the factors of a word that have a given period, we improve on the results of [16] dealing with algorithms and data structures that help us answer in constant time queries regarding the periodicity of the factors of a word. Whenever the words are extended by the simple operation of adding one symbol (that is, a regular letter or a hole) at the right end of the word, we can update the data structures we constructed in linear time such that the query-answering time remains unchanged. Note that the idea of updating a word by adding a letter at its end while managing information regarding its combinatorial properties (initiated in [16] and developed in [1]) opens the discussions on streaming and on-line algorithms testing combinatorial properties of partial words (and their factors).

Second, in Section 4, we give optimal algorithms that identify ways of making a word periodic by substituting the letters on some positions of the input word by holes in a restricted manner and improve the already mentioned results from [15]. While the results of Section 3 regard some natural algorithmic questions on the periodicity of words, note that the results presented in Section 4 may become useful in the area of combinatorics on words. For instance, we might be interested in constructing words in which we can randomly substitute letters with holes, such that no two holes are too close, while they remain nonperiodic ([14, 15]). Theorem 4, from the Section 4, enables us to test efficiently whether a given word satisfies this property or not. This strategy is not new, as in [17, 18] the idea of producing from given full words, by substitution of some positions of the words with holes, partial words that satisfy some combinatorial properties is discussed, and possible connections with bioinformatics are established. Furthermore, Theorem 4 seems important as it shows the existence of an optimal algorithm used in a meaningful context (to prove [15, Conjecture 1]).

2. Basic Definitions

We continue with several basic definitions.

Let V be a non-empty finite set of symbols called an *alphabet*, and denote its cardinality by $|V|$. Each element $a \in V$ is called a *letter*. A *full word* over V is a finite sequence of letters from V. A *partial word* over V is a finite sequence of letters from $V_\diamond = V \cup \{\diamond\}$, the alphabet V extended with the hole symbol $\diamond$. A full word is a partial word that does not contain the $\diamond$ symbol. The set containing all finite full words over the alphabet V is denoted by V^*, while the set of all finite partial words over the alphabet V is denoted by $V_\diamond^*$.

The *length* of a partial word u is denoted by $|u|$ and represents the total number of symbols in u. The *empty word*, denoted by ε, is the sequence of length zero. Thus, alternatively, a length n partial word $u \in V_\diamond$ can be viewed as a function $u : \{1, \ldots, n\} \to V_\diamond$ or as a partial function $u : \{1, \ldots, n\} \overset{\circ}{\to} V$.

A partial word u is a *factor* of a partial word v if $v = xuy$ for some x, y. We say that u is a *prefix* of v if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$. We denote by $u[i]$ the symbol at position i in u and by $u[i..j]$ the factor of u starting at position i and ending at position j, consisting of the concatenation of the symbols $u[i], \ldots, u[j]$, where $1 \leq i \leq j \leq n$.

If u and v are partial words of equal length, then u is *contained* in v, denoted $u \subset v$, if $u[i] = v[i]$, for all $u[i] \in A$. Moreover, partial words u and v are *compatible*, denoted by $u \uparrow v$, if exists w such that $u \subset w$ and $v \subset w$.

The powers of a partial word u are defined recursively by $u^0 = \varepsilon$ and for $n \geq 1$, $u^n = uu^{n-1}$. A *period* of a partial word u over V is a positive integer p such that $u[i] = u[j]$ whenever $u[i], u[j] \in V$ and $i \equiv j \pmod{p}$. In such a case, we say u is *p-periodic*. If $p < |u|$, then u is periodic. A partial word u is said to be a k-repetition if it has a period p such that $p = \frac{|u|}{k}$; if the partial word u is not a k-repetition for any $k > 1$ we say that u is primitive.

As an example for the above, we see that the length 6 partial word $w = ab\diamond ba\diamond$ is 2-periodic and, thus it is a 3-repetition, since $a\diamond, \diamond b \subset ab$, and, therefore, $ab \uparrow \diamond b$, $ab \uparrow a\diamond$, and $\diamond b \uparrow a\diamond$.

A partial word u is said to be d-valid for some positive integer d if $u[i..i+d-1]$ contains at most one $\diamond$-symbol, for all i with $1 \leq i \leq |u| - d + 1$. For the above example, we see that w is d-valid for any $d \in \{1, 2, 3\}$.

For a complete view on the basic definitions regarding combinatorics on words and partial words we refer the reader to [2, 4].

The basic definitions needed in order to follow the algorithms presented here are found in the handbook [19]. We stress that all the time bounds provided in this paper hold on the *unit-cost RAM (Random Access Machine) with logarithmic word size* model. For a more detailed description of this model, [19, Section 2.2] is a good reference. In this model (which is generally used in the analysis of algorithms) we assume that for an input of length n each memory cell can store $\mathcal{O}(\log n)$ bits, or, in other words, that *the machine word size* is $\mathcal{O}(\log n)$; the constant hidden by the $\mathcal{O}$-notation is at least 1. The instructions are executed one after another, with no concurrent operations. The model contains common instructions: arithmetic (add, subtract, multiply, divide, remainder, bitwise shifts), data movement (load the content of a memory cell, store a number in a memory cell, copy the content of a memory cell to another), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time. As it is important in our paper, we point out that testing the equality of two numbers is also assumed to take a constant amount of time; as a consequence, we note that also testing whether a number with $\mathcal{O}(\log n)$ bits divides another number with $\mathcal{O}(\log n)$ bits takes constant time, as well. It is also a common assumption, made when working with the unit-cost RAM with logarithmic word size, that basic operations on arrays (like accessing or updating the values found at a certain position of the array) containing a polynomial (in n) number of $\mathcal{O}(\log n)$-bit integer elements, are carried out in constant time. Basically, this model allows us to measure the number of instructions executed in an algorithm, making abstraction of the time spent to execute each of the basic instructions.

As an addition to the above comments, it is worth considering the case when the only arithmetic operations that the computational model we use can execute in constant time are addition, subtraction, multiplication, but not division. In this case, using the Newton–Raphson division method, we obtain the result of the integer-division of two $\mathcal{O}(\log n)$ bits integers in $\mathcal{O}(\log \log n)$

time (that is, this many executions of basic steps). Therefore, detecting all the divisors of a number n, that has $\mathcal{O}(\log n)$ bits, can be done canonically in $\mathcal{O}(n(\log\log n))$ time. Moreover, we can precompute, using the Sieve of Eratosthenes, all the pairs (k,ℓ) such that k divides ℓ and $k,\ell \leq n$ in $\mathcal{O}(n\log n)$ time (and create a very simple look up table for them in $\mathcal{O}(n^2)$ time). Building upon these remarks, we have that, even for this more general model, all the algorithmic results we present (especially those in Section 3, where division plays an important role) still hold.

Last, but not least, in the algorithmic problems we discuss, when given an input partial word w of length n we assume that all its letters are from the set $\{0,1,\ldots,n\}$ with 0 encoding the hole. Consequently, w is seen as a sequence of integers. In this setting, basic operations on the symbols of partial words can be performed in constant time, in our computational model. Note that such assumptions are rather usual in algorithmic problems on words (see, e.g., the discussion in [20]).

3. Testing the periodicity of partial words

We start this section by presenting a series of algorithms that efficiently identify all the periods of a given partial word. Next, we follow an approach from [16] and discuss how we can efficiently build data structures that allow us to answer in constant time queries asking whether a factor of a given partial word is periodic, or which is the minimum period of a partial word. Moreover, we also present a method to update these data structures whenever new symbols are added to the initial word.

3.1. Finding all the periods of a word

First, let us remark that it is trivial to check in linear time if a given partial word is p-periodic, when p is also given as input.

On the other hand, testing if an input partial word w of length n is periodic can be done in $\mathcal{O}(n\log n)$ time by the following quite simple algorithm. We observe that the partial word w over V is periodic if and only if it is compatible with a word uvu with $u,v \in V^*$ and $|u|>0$. But w is compatible with a word of form uvu, as above, if and only if w is compatible with a factor of the word $w[\lfloor\frac{n}{2}\rfloor+1..n]\diamond^{n-1}$. This is checked in $\mathcal{O}(n\log n)$ time using the pattern matching algorithm proposed in [11]. Therefore, if $w[i..n]\diamond^{i-1}$ is the rightmost factor of $w[\lfloor\frac{n}{2}\rfloor+1..n]\diamond^{n-1}$ compatible with w, the longest period of w is in this case $i-1$.

Further, we provide an algorithm that finds all periods of a given partial word w, of length n. We start by an observation:

Remark 1. *A word w with $|w| = n$ is p-periodic if and only if, for all integers i with $0 < ip < n$, the length n prefix of $w[ip+1..n]\diamond^n$ is compatible with w.*

To see this, assume that $n = qp + r$ with $0 \leq r < p$. Indeed, if w is p-periodic, then it is clear that for all i, such that $1 \leq i \leq q$, we have that $w[1..(q-i)p]$ is compatible with $w[ip+1..qp]$, $w[(q-i)p+1..(q-i)p+r]$ is compatible with $w[qp+1..n]$, and, finally, $w[kp+r+1..n]$ is compatible with $\diamond^{n-kp-r}$. This concludes the direct implication. For the reverse implication, observe that, since w is compatible with the length n prefix of each of the words $w[ip+1..n]\diamond^n$, where i is a non-negative integer with $ip < n$, it follows that $w[jp+1..(j+1)p]$ is compatible with $w[j'p..(j'+1)p]$ and $w[jp+1..jp+r]$ is compatible with $w[qp+1..n]$, for all integers j and j' such that $1 \leq j, j' \leq q$. Following this, we easily obtain the existence of a length p full word u that is compatible with $w[jp+1..(j+1)p]$ for all integers j with $0 \leq j \leq q$, and $u[1..r]$ compatible with $w[qp+1..n]$. Actually, this implies that w is p-periodic.

The above remark suggests the following approach for finding the periods of a word. As a first step, we find all the integers i with $1 \leq i \leq n$, such that w is compatible with the factor $w[i..n]\diamond^{i-1}$ of $w\diamond^n$. We store an n positions array *Occ* where $Occ[i] = 1$ if w is compatible with $w[i..n]\diamond^{i-1}$ and $Occ[i] = 0$, otherwise. The second step consists in identifying the positive numbers p that fulfil $Occ[kp+1] = 1$, for all integers k such that $0 \leq kp \leq n$. As we have previously noted, such a number p is a period of w.

The first step of the algorithm can be completed in $\mathcal{O}(n \log n)$ time by running the pattern matching algorithm from [11] in order to find all the occurrences of w in $w\diamond^n$.

The second step is implemented even faster, in $\mathcal{O}(n(\log\log n))$ time, but in a more involved manner. As main idea, we identify all the numbers p such that at least one of the values of $Occ[kp+1]$ equals 0, for k such that $0 \leq kp \leq n$. To find these numbers, we note that if an integer i is not a period of w then not all the numbers $\frac{i}{p}$, for p a prime divisor of i, are periods of w; further, we apply this reasoning recursively for the numbers $\frac{i}{p}$, where p is a prime divisor of i.

To implement the above strategy, we compute and store for every i with $1 \leq i \leq n$ its prime divisors. This is done as follows. Using the Sieve of Eratosthenes we find all the primes less than or equal to n, in time $\mathcal{O}(n(\log\log n))$. In the same process, we compute for each i with $1 \leq i \leq n$ a list $L[i]$ of the prime numbers that divide it. The total number of primes that these lists contain is in $\mathcal{O}(n(\log\log n))$. Indeed, a prime p divides at most $\frac{n}{p}$ numbers less than or equal to n, so it will appear in $\left\lfloor \frac{n}{p} \right\rfloor$ lists. Therefore, the number of primes contained in the lists $L[i]$ with $1 \le i \le n$ is

$$\sum_{p \text{ prime}, \, p \le n} \left\lfloor \frac{n}{p} \right\rfloor \;\le\; \sum_{p \text{ prime}, \, p \le n} \frac{n}{p} \;=\; n \cdot \sum_{p \text{ prime}, \, p \le n} \frac{1}{p} \;\in\; \mathcal{O}(n(\log\log n)).$$

Once we finish computing the lists $L[i]$ for $1 \le i \le n$ we define an array Per with $n-1$ elements and initially set $Per[i] = 1$ for all $i \in \{1, \ldots, n-1\}$. Then, for all values i, in decreasing order, from $n-1$ to 0, if $Occ[i+1] = 0$ or $Per[i] = 0$ we set $Per[j] = 0$ for all the elements $j = \frac{i}{p}$ where $p \in L[i]$ (that is, for every j that has the form $\frac{i}{p}$ with p a prime divisor of i). Clearly, computing the array Per takes $\mathcal{O}(n(\log\log n))$ time and from $Per[i] = 1$ it follows that $Per[ki] = 1$, for all integers k such that $1 \le ki + 1 \le n$.

According to the above remarks, a number p is a period of w if and only if $Per[p] = 1$. Consequently, all periods of w are computed in $\mathcal{O}(n \log n)$ time. We stress that the most time consuming part is the identification of the numbers i such that $w[i..n]\diamond^{i-1}$ is compatible with w.

These results are particularly useful in two applications: finding the minimal period of a partial word and deciding whether a word is primitive. We are aware of the claims and proofs that these problems can be solved in linear time (see [10] and, respectively, [9]). However, the algorithms proposed in these papers rely on the fact that we can find all factors of a partial word ww that are compatible with w in linear time, by extending to the case of partial words some linear time string matching algorithms for full words. The proof of this fact was not given formally, and we are not convinced that such extensions actually hold, especially since the compatibility relation is not transitive (e. g., see the discussions in [12]) and, also, since for partial words the length does not always equal the sum between the period and the border of the word as in the case of regular words. For this last fact, consider the word $a\diamond b$ that has a border of length 2 and is aperiodic (for more details we refer to [21]).

Moreover, our algorithm enables us to find the smallest period of a partial word w (find the minimum p with $Per[p] = 1$), as well as decide if a partial word w is primitive (check if there exists a p such that $Per[p] = 1$ and p divides $|w|$), both tasks within $\mathcal{O}(n \log n)$ running time.

The results of this section are expressed by the following Theorem:

Theorem 1. *All periods of a partial word w with $|w| = n$ can be computed in time $\mathcal{O}(n \log n)$.*

3.2. Queries and Updates

The problem we discuss in this section is strongly related to the fundamental problem of avoidability, or freeness, of (partial) words. Therefore, we consider natural finding the repetitions within a partial word, or, in an even simpler context, the periodic factors of such a word. It is worth mentioning that identifying repetitive structures inside words is strongly motivated by the problem of detecting repeated regions (with errors, modelled here by the holes) in a DNA or protein sequence [22].

We propose an approach allowing us to treat all such problems in a uniform manner. More precisely, we are interested to construct data structures that enable us to answer in constant time queries regarding the periodicity of the factors of a given partial word, and, moreover, are easily updated once the word is extended. We formalise this as the following problem:

Problem 1. *Let w be a partial word of length n defined over an alphabet V.*
1. *Preprocess w in order to answer for integers $i, j, p \in \{1, \dots, n\}$ with $i \leq j$ the following types of queries:*

"Is $w[i..j]$ p-periodic?", denoted $\mathbf{per}(i, j, p)$.

"Which is the minimum period of $w[i..j]$?", denoted $\mathbf{minper}(i, j)$.

2. *Consider the update operation:* "add $a \in V_\diamond$ to the right end of w, to obtain wa". *Preprocess w and define a method to update the data structures constructed during the preprocessing phase, in order to answer in constant time* **per** *queries, for a word obtained after several update operations on w.*

Before providing a solution to this problem, let us highlight several of its immediate applications. For instance, for a given p we can use **per** queries to identify in quadratic time all the pairs (i, j) such that $w[i..j]$ is p-periodic. On the other hand, we can identify for a given p all the pairs (i, j) such that the minimum period of $w[i..j]$ is greater (respectively, smaller) than p, using, in this case, **minper** queries. To this end, Remarks 2, 3, and 4 will show how to use the results obtained during the analysis of Problem 1 to identify all the periodic, or primitive, factors of a partial word, as well as the maximal periodic factors (in the sense that appending or prepending a letter to these factors produces non-periodic partial words), or maximal periodic factors whose smallest period is at most half of the length of the factors (similar to runs, in the case of full words).

Finally, note that in order to be able to implement the update operations efficiently, we need to recall several facts on standard data structures. It is known (e. g., see the handbook [19]) that adding a new element to an array can be done in constant amortised time, or linear time in the worst case, and the access time to the elements of the array remains constant. From this it follows (see [16, Section 3]) that extending matrices by appending a column or a row to them can be implemented in $\mathcal{O}(1)$ amortised time complexity and linear time in the worst case, while the access time to elements of the (updated) matrix remains constant. Consequently, adding both a row and a column to an $m \times n$ matrix requires $\mathcal{O}(n+m)$ time in the worst case, and the access time is preserved. We stress that the matrix manipulation and update techniques proposed in [16, Section 3], which we reference and use here, do not change, conceptually, the way we work with the elements of a matrix at all.

3.2.1. Algorithms

We begin by describing a solution for the first part of Problem 1, that can be easily adapted to solve also its second part.

First define the $n \times n$ matrix A, where for $i, l \in \{1, \ldots, n\}$:

$$A[i][l] = \begin{cases} \max\{k \mid 0 < k \leq i, i-k \text{ is divisible by } l \text{ and } w[k] \neq \diamond\}, \\ \qquad \text{if the set is non-empty;} \\ i - \lfloor (i-1)/l \rfloor l, \text{ otherwise.} \end{cases}$$

Basically, $A[i][l]$ equals k, where $0 < k \leq i$ and $i-k$ divisible by l if all the symbols $w[k+jl]$ with $1 \leq j \leq \frac{i-k}{l}$ are equal to $\diamond$ and $w[k] \neq \diamond$, or, if such an integer k does not exist, $A[i][l]$ equals the leftmost position t of the word with $i-t$ divisible by l. This matrix can be computed in $\mathcal{O}(n^2)$ time by dynamic programming:

$$A[i][l] = \begin{cases} i, \text{ if } w[i] \neq \diamond \text{ or } i \leq l; \\ A[i-l][l], \text{ otherwise.} \end{cases}$$

To see that the formula holds, note that if $w[i] \neq \diamond$ or $i \leq l$, then $A[i][l] = i$. If none of these conditions hold, then $A[i][l] = A[i-l][l] = k \neq i$. Indeed, if $w[k] = \diamond$, then $w[k+tl] = \diamond$, for all $t \in \{0, \ldots, (i-k)/l\}$, and, since $i > l$, we have $A[i-l][l] = k$ as well. If $w[k] \neq \diamond$, then $w[k+tl] = \diamond$, for all $t \in \{1, \ldots, (i-k)/l\}$. Thus, $A[i-l][l] = k$ also, since $w[k+tl] = \diamond$, for every $t \in \{1, \ldots, (i-l-k)/l\}$.

Now define the $n \times n$ matrix T, where for $i, l \in \{1, \ldots, n\}$:

$$T[i][l] = \min\{j \mid 0 < j \leq i, \text{ such that } w[j..i] \text{ is } l\text{-periodic}\}.$$

We make a series of remarks regarding T, that hold for all i and l:

- $T[i][l] = j > 1$ if and only if $w[j..i]$ is l-periodic and $w[j-1..i]$ is not.
- $T[i][l] = 1$ if $i \leq l$, since $w[1..i]$ is clearly l-periodic. Moreover, if $i > l$, then $w[i-l+1..i]$ is l-periodic and $i - T[i][l] + 1 \geq l$. As a consequence we have $T[1][l] = 1$ and $i > T[i][l]$ whenever $i > 1$.
- $T[i][l] \geq T[i-1][l]$ if $i > 1$. Indeed, if $w[p..i]$ is l-periodic, then $w[p..i-1]$ is also l-periodic. If $T[i][l] = T[i-1][l]$, then $i - T[i][l] + 1 > l$.
- the above remarks show that $i - T[i-1][l] + 1 > l$ if $i > l$.
- for $t = T[i-1][l]$ and $i > l$, following the above, we have $i - l \geq t$ and:

  – if $A[i-l][l] < t$, then $w[t..i]$ is periodic and $w[t-1..i]$ is not, since $w[j] = \diamond$ for all $i - j$ divisible by l and $i > j \geq t$, and $T[i][l] \geq t$. As a consequence $T[i][l] = T[i-1][l]$.

  – if $A[i-l][l] \geq t$ and $w[i] \uparrow w[A[i-l][l]]$, then all symbols $w[j]$ with $i - j$ divisible by l and $i \geq j \geq t$ are contained in the same symbol. Thus, $w[t..i]$ is l-periodic. As a consequence $T[i][l] = T[i-1][l]$.

  – if $A[i-l][l] \geq t$ and $w[i] \not\uparrow w[A[i-l][l]]$, then $w[j] \subset w[i]$ for all $i - j$ divisible by l and $i \geq j > A[i-l][l]$, and $w[A[i-l][l]+1..i]$ is l-periodic, while $w[A[i-l][l]..i]$ is not l-periodic. Thus, $T[i][l] = A[i-l][l] + 1$.

These remarks show that a matrix T can be computed by dynamic programming using the following relations that hold for all $i, l \in \{1, \ldots, n\}$. Whenever $i \leq l$, we have $T[i][l] = 1$. Otherwise:

$$T[i][l] = \begin{cases} T[i-1][l], & \text{if } A[i-l][l] < T[i-1][l]; \\ T[i-1][l], & \text{if } A[i-l][l] \geq T[i-1][l] \text{ and } w[i] \uparrow w[A[i-l][l]]; \\ A[i-l][l] + 1, & \text{otherwise.} \end{cases}$$

Hence, the time needed to compute the elements of the matrix T is $\mathcal{O}(n^2)$. Now, we use matrix T to answer **per** queries, as described in Algorithm 1.

Next we describe the way to deal with **minper** queries. We define the matrix P_m with n rows and n columns, that stores the answer to our queries. Thus for $i, j \in \{1, \ldots, n\}$ with $i \leq j$ we have:

$$P_m[i][j] = \min\{p \mid w[i..j] \text{ is } p\text{-periodic}\}.$$

Algorithm 1 Periodic: answering **per** queries for the input word w.

[Preprocessing:] Construct the matrices A and T for the word w, as previously described.

[Query:] The answer to a query $\mathbf{per}(i, j, p)$ is **yes** if $T[j][p] \leq i$ and **no**, otherwise.

Hence if $i > j$, we have $P_m[i][j] = 0$. Moreover, if $w[i..j]$ is not p-periodic, for any $p < j - i + 1$, then $P_m[i][j] = j - i + 1$. Also, $P_m[i][j] \geq P_m[i+1][j]$. Consequently, the matrix P_m can be computed using Algorithm 2.

Algorithm 2 Compute P_m: how to compute **minper** for all factors of w.

```
 1: construct the matrices $A$ and $T$ for the word $w$, as previously described.
 2: for $1 \leq j < i \leq n$ set $P_m[i][j] = 0$.
 3: for $1 \leq i \leq j \leq n$ set $P_m[i][j] = j - i + 1$.
 4: for $j = 1$ to $n$ do
 5:    set $l = j$.
 6:    for $p = 1$ to $n$ do
 7:       if ($T[j][p] = j'$ and $l > j'$) then
 8:          for $i = j'$ to $l$ do set $P_m[i][j] = p$.
 9:          set $l = j'$.
10:    end for
11: end for
```

In other words, for an integer j with $1 \leq j \leq n$ we first identify the longest factor $w[i_1..j]$ of $w[1..j]$ that is 1-periodic, and conclude that the minimum period of $w[\ell..j]$ is 1 for $i_1 \leq \ell \leq j$. Next, we identify the longest factor $w[i_2..j]$ of $w[1..j]$ that is 2-periodic, and conclude that the minimum period of $w[\ell..j]$ is 2, where $i_2 \leq \ell \leq i_1 - 1$. The process continues for all periods p with $3 \leq p \leq j$ by identifying the longest p-periodic factor $w[i_p..j]$ of $w[1..j]$ and concluding that the minimum period of $w[\ell..j]$ is p, where $i_p \leq \ell \leq i_{p-1} - 1$.

The time complexity for computing P_m, by Algorithm 2, is $\mathcal{O}(n^2)$. First, the construction of the matrices A and T is done in quadratic time, as already shown. The initialisation of P_m takes $\mathcal{O}(n^2)$ time, as well. Next, for a fixed j, the complete execution of the cycle in step 6 takes $\mathcal{O}(n)$ time. Indeed, in this cycle we set the values on one of the columns of matrix P_m. There are n such values, so we need exactly n operations since we do not set an element of the matrix more than once. Consequently, the entire cycle 4 requires $\mathcal{O}(n^2)$ time, which shows the overall quadratic complexity of the algorithm.

While Algorithm 2 provides the setting, Algorithm 3 depicts the way **minper** queries are answered.

Algorithm 3 MinPer: answering **minper** queries for the input word w.

```
[Preprocessing:] Construct for $w$ the matrices $A$, $T$ and $P_m$, as previously described.
[Query:] The answer to a query $\mathbf{minper}(i, j)$ is $P_m[i][j]$.
```

It is worth noting that in the case of answering **per** queries we produced data structures that do not contain exactly the answer to every possible query, but help us compute this answer in constant time. On the other hand, in the case of **minper** queries we produced an oracle-structure, the matrix P_m, that already contains the answers to all the possible queries, and this answers can be retrieved in constant time.

Let us now describe how to solve the second part of Problem 1 when exactly one update is applied to a given partial word. If the word is updated more than once, we simply iterate this method. Suppose that we are given a length n partial word w, for which we compute the matrices A and T, as previously described. Further, assume that we add to w the symbol $a \in V_\diamond$ and obtain the partial word $w' = wa$, where $w'[n + 1] = a$. Update the matrices A and T, by adding to each of them a new column and a new row. We observe that $A[i][n+1] = i$ and $T[i][n+1] = 1$, for all $i \in \{1, \ldots, n+1\}$, while the elements $A[n + 1][l]$ and $T[n + 1][l]$ can be computed using the already described recurrence-formulas, for all $l \in \{1, \ldots, n\}$. Moreover, we add the new rows and columns to the matrices T and A in time $\mathcal{O}(n)$, while the computation of the newly inserted elements also requires $\mathcal{O}(n)$ time. Once all structures are updated, we answer **per** queries exactly as described at the second step of Algorithm 1.

In order to update the matrix P_m, once A and T are updated, we first set $P_m[n+1][n+1] = 1$ and $P[n+1][j] = 0$, for all $j \leq n$. Then, we run the cycle from step 6 of Algorithm 2 for $j = n + 1$, in order to compute the elements $P[i][n + 1]$ with $i \leq n$. This clearly takes $\mathcal{O}(n)$ time. Of course, the answer to **minper** queries is still obtained as in the second step of Algorithm 3.

To summarise, we are able to show the following result:

Theorem 2. *A length n partial word w can be processed in time $\mathcal{O}(n^2)$ in order to answer* **per** *and* **minper** *queries, in time $\mathcal{O}(1)$. If update operations, in which a new symbol is added to the rightmost end of w, are applied, the previously constructed data structures are updated in at most $\mathcal{O}(n)$ time per update, and both* **per** *and* **minper** *queries are still answered in time $\mathcal{O}(1)$.*

This represents an improvement of the results presented in [16]. In that paper the preprocessing time needed to answer in constant time queries asking if a factor of a word is a k-repetition, k-free or overlap-free where k is a positive integer is $\mathcal{O}(n^2)$, while the time needed for updating the data structures is $\mathcal{O}(n \log n)$ in a worst case analysis and $\mathcal{O}(n)$ in an amortised analysis. It is easy to see that the queries asking whether a factor of the word is a repetition are particular cases of **per** queries, so they can be answered in $\mathcal{O}(1)$ time using the data structures constructed for Problem 1. Further, as shown in [16], all the other types of mentioned queries can be answered in constant time by asking whether factors of the word are repetitions. Thus, following the reasoning in the aforementioned paper, we easily obtain that the preprocessing time needed to answer in constant time queries asking whether a factor of a word is overlap-free, a k-repetition, or k-free, for some positive integer k given in the query, is $\mathcal{O}(n^2)$, while the time needed for updating these data structures is $\mathcal{O}(n)$ in the worst case. Furthermore, the data structures used here are quite simple ones.

We end this section with a few observations.

Remark 2. *The matrix T, computed for a partial word w, can also be used to determine all periodic factors of the word in quadratic time. The maximal periodic factors of a word can be also computed in quadratic time.*

We compute for each j the minimum i such that $w[i..j]$ is p-periodic for some $p < j-i+1$ and store this value as $min[j]$. The array min is computed in quadratic time from the matrix T by looking at the minimum of the set $P_j = \{T[j][l] \mid l \leq n\}$, which is done in linear time. Clearly, a factor $w[i'..j']$ is periodic if and only if $min[j'] \leq i' \leq j'$, which is checked in constant time.

Once all the periodic factors of w are computed, one can easily detect the maximal periodic factors of w. We just have to compute for each i the maximum j such that $w[i..j]$ is p-periodic for some $p < j - i + 1$ and store this value as $max[j]$. Basically, $w[i..j]$ is a maximal periodic factor, if and only if $min[j] = i$ and $max[i] = j$.

However, runs cannot be defined as in the case of full words. In the case of partial words, there might be factors $w[i..j]$ of a word w having two periods p_1 and p_2, both smaller than half the length of $w[i..j]$, and extending $w[i..j]$ by even only one letter may lead to a word that is not p_1-periodic, but remains p_2-periodic. For example, take $w = aba\diamond aba\diamond abacdd$; then $w[1..11]$ is both 2-periodic and 4-periodic. However, $w[1..12]$ is not 2-periodic but it is 4-periodic, so, indeed, $w[1..11]$ can be extended to a factor that is periodic with respect to one of its periods but non-periodic with respect to another; finally, $w[1..13]$ does not have a period less or equal to 6. Clearly, the Theorem of Fine and Wilf (see [4]) ensures that such situations are not possible in the case of full words.

Therefore, let us define the runs of a partial word w as the factors $w[i..j]$ of w which are p-periodic, for some $p \leq \frac{j-i+1}{2}$, and both $w[i-1..j]$ and $w[i..j+1]$ (if these words can be defined) are not p'-periodic, for any $p' \leq \frac{j-i+2}{2}$. Looking back at our example, $w[1..11]$ is not a run, while $w[1..12]$ is a run.

Remark 3. *The matrix P_m helps us detect all runs in a partial word in quadratic time.*

The factor $w[i..j]$ is a run if and only if we have that $P_m[i][j] \leq \frac{j-i+1}{2}$, $P_m[i-1][j] > \frac{j-i+2}{2}$ (when $i-1 > 0$), and $P_m[i][j+1] > \frac{j-i+2}{2}$ (when $j+1 < n+1$). Hence, again one can decide in constant time whether a factor $w[i..j]$ is a run or not. Finding all the runs in the word is done, clearly, in $\mathcal{O}(n^2)$ time.

Remark 4. *The data structures constructed to answer* **per**-*queries enable us to compute the primitive factors of w in $\mathcal{O}(n^2 \log\log n)$ time.*

The factor $w[i..j]$ is primitive if and only if for every prime divisor p of $\ell = j-i+1$ we have that $\mathbf{per}(i, j, \frac{\ell}{p})$ returns **no**, property which can be checked in constant time per prime divisor, once the data structures needed to answer **per**-queries are constructed. As we can compute the lists of prime divisors of all $\ell \leq n$ in $\mathcal{O}(n \log\log n)$ time, using the Sieve of Eratosthenes, if we implement the check for primitive factors of w by considering in increasing order of the length these factors, then the overall time complexity of this task is clearly $\mathcal{O}(n^2 \log\log n)$.

4. From full words to periodic partial words

In this section we change our focus to constructing, in linear time, a p-periodic partial word starting from a full word, by replacing some of its symbols with holes such that no two consecutive holes are too close one to the other. This is formally expressed by the following problem:

Problem 2. *[15] Given a word $w \in V^*$, and positive integers d and p, both less than $|w|$, decide whether there exists a p-periodic d-valid partial word contained in the word w.*

In order to present a solution for Problem 2 we first show how another strongly related problem is solved in linear time. Observe that, given a word w we can obtain a sequence of arrays M by putting in the array M_i the letters $w[kp+i]$, for $k \geq 0$. Now, the p-periodicity and d-validity conditions defined for words can be expressed based on the properties of the arrays M.

Problem 3. *Let $M = M_1, \ldots, M_m$ be a sequence of m arrays over V with at most q elements each, and d be a positive integer with $d \leq m$. Furthermore, let $k \geq 1$ be a positive constant, and $p_0, \ldots, p_k \in \{0, \ldots, m\}$ such that $p_0 = 0$, $p_k = m$ and $p_i < p_{i+1}$, for $i \in \{0, \ldots, k-1\}$; assume that the arrays $M_{p_i+1}, \ldots, M_{p_{i+1}}$ have $q-i$ elements. Decide whether we can replace positions of M_i by holes in order to obtain arrays M'_i, where $i \in \{1, \ldots, m\}$, such that the following two conditions hold:*

1. *There exists a symbol s_i that contains all the symbols of M'_i with $i \in \{1, \ldots, m\}$. This condition is called* **the periodicity condition.**
2. *For all integers $j \leq q$ and $i \in \{1, \ldots, m-d+1\}$, there exists at most one hole in a multiset $\{M'_i[j], \ldots, M'_{i'}[j]\}$ where $i' = \max\{i+d-1, p_{q-j+1}\}$. This condition is called* **the d-validity condition**.

The input consists of the sequence M and the number d. In this case, following the discussions from Section 2, we can assume that $|V| \leq mq$ and we need $\mathcal{O}(mq)$ memory words to store the input. Observe that, the conditions 1 and 2 are reformulations in this setting of the p-periodicity and d-validity conditions defined for words. The precise relation between Problem 2 and Problem 3 and the way we can solve the former problem by solving the latter one are discussed in Section 4.3.

We explain a little more the d-validity condition above. This condition requires that for each possible position $j \leq q$ and each $i \leq m-d+1$, if we take the d arrays $M'_i, \ldots, M'_{i+d-1}$ and select from them the ones that have at least j elements (that is, whose index is at most p_{q-j+1}), then there is at most one hole occurring on the j^{th} position of these arrays.

4.1. A solution for Problem 3

Intuitively, the solution of this problem is based on the following idea: a possible way to substitute symbols of an array with holes, in order to fulfil the periodicity condition, induces some restrictions on the way the substitutions can be applied on the $d-1$ arrays that follow it (due to the d-validity condition). Thus, we obtain a set of restrictions for each d consecutive arrays. We propose a formalisation of the substitutions on each array as boolean variables and of the restrictions induced by these substitutions as formulas involving the variables. The fact that all the restrictions must be simultaneously fulfilled is reflected by the satisfiability of the conjunction of all the formulas. We show how the formulas are constructed and how the satisfiability of their conjunction is efficiently decidable.

Before describing the solution of Problem 3, let us define the sequence of arrays M^b with elements from $\{0,1\}$ as follows: $M_i^b[j]=1$ if $M_i[j]=M_i[1]$, and $M_i^b[j]=0$, otherwise. Note that, there are two types of substitutions (also called replacements) that we can apply to the symbols of an array M_i in order to be sure that there exists a symbol $s_i \in V$ that contains all the elements of the modified array:

Type* 1*: We replace with holes all the symbols $M_i[j]$ different from $M_i[1]$. In this case $M_i'[j]=\diamond$ for all j such that $M_i^b[j]=0$, while all the other positions are equal to $M_i[1]$.

Type* 2*: There exists a symbol $M_i[j_0]$, different from $M_i[1]$, such that we can replace with holes all the symbols $M_i[j]$ different from $M_i[j_0]$. In this case $M_i'[j]=\diamond$ for all j such that $M_i^b[j]=1$ and all the symbols of the array are now contained in $M_i[j_0]$.

We see that for a given array no substitution is necessary in the case when all of its symbols are already equal. By the d-validity condition, the following remarks are straightforward:

Remark 5. *In d consecutive arrays, only one* Type 2 *replacement is possible.*

Remark 6. *If for some k with $1 \le k \le q$ we have $M_{j_0}^b[k]=M_{j_1}^b[k]=0$ where $0<j_1-j_0\le d$, then we cannot apply* Type 1 *replacements to both M_{j_0} and M_{j_1}.*

After these basic remarks let us solve Problem 3.

To begin with, define the boolean variables x_i for $i \in \{1,\ldots,m\}$ which are true, i.e., $x_i=1$, if and only if we apply a *Type 2* replacement to M_i in order to reach a valid solution for the problem. Then, for each d consecutive arrays of M, say $M_i, \ldots, M_{i+d-1}$, we construct a boolean formula, denoted ϕ_i, involving some of the variables x_j defined above, with $i \leq j \leq i+d-1$. The formal definitions for each ϕ_i are given in Algorithm 4 and explained in the case analysis preceding this algorithm. All these formulas are the conjunction of at most 2 clauses. Intuitively, the formula ϕ_i can be satisfied if and only if some of the symbols of the corresponding arrays can be replaced with holes and obtain a sequence that satisfies both the periodicity and the d-validity conditions. Finally, we derive from all these small formulas a new formula ϕ_M, and show that there exists an assignment of the variables x_i with $i \in \{1, \ldots, m\}$ that makes ϕ_M true if and only if Problem 3 has a solution. A solution for Problem 3 is derived according to the truth values of the variables x_i, where $i \in \{1, \ldots, m\}$.

In the following we show how this intuition can be formally implemented, such that the algorithm runs in $\mathcal{O}(mq)$ time. The time complexity does not depend on $|V|$ nor on the value of d.

First, we consider d consecutive arrays $M_i, \ldots, M_{i+d-1}$, all with the same number of elements q. Also, we assume $q \geq 2$, as the case when $q = 1$ is trivial, since no replacement needs to take place.

Remark 7. *If for some k with $1 \leq k \leq q$, we have $M^b_{j_1}[k] = M^b_{j_2}[k] = M^b_{j_3}[k] = 0$, where $j_1 < j_2 < j_3$ and $j_3 - j_1 < d$, we cannot obtain a sequence that satisfies both the periodicity and the d-validity condition, no matter the replacements we apply.*

Indeed, according to Remark 5 we can apply only one *Type 2* and two *Type 1* replacements to these three arrays. Making use of Remark 6, we get that the problem has no solution.

Similar arguments show that in order to apply a *Type 2* replacement to an array M_j where $j \in \{i, \ldots, i+d-1\}$ the following conditions must hold:

- if there exist k with $1 \leq k \leq q$ and $j_1, j_2 \in \{i, \ldots, i+d-1\}$ with $j_1 < j_2$ such that $M_{j_1}[k] = M_{j_2}[k] = 0$, then $j \in \{j_1, j_2\}$.
- if there exist k with $1 \leq k \leq q$ and a unique $j_1 \in \{i, \ldots, i+d-1\}$ such that $M_{j_1}[k] = 0$, then $j = j_1$.

Further, suppose we apply a *Type 2* replacement to the array M_j for some $j \in \{i, \ldots, i+d-1\}$ and there exists k such that $M^b_j[k] = 1$. Then $M^b_\ell[k] \neq 0$, for all $\ell \in \{i, \ldots, i+d-1\}$. Indeed, if an ℓ exists such that $M^b_\ell[k] = 0$, then according to Remark 5 we can only apply a *Type 1* replacement to M_ℓ.

However, after these two substitutions $M'_j[k] = M'_\ell[k] = \diamond$, a contradiction with the d-validity condition.

Now, for an integer k with $1 \le k \le q$ let:

$$L_i(k) = \begin{cases} \{j \mid j \in \{i, \ldots, i+d-1\}, M^b_j[k] = 0\}, \text{if the set is non-empty,} \\ \{1, \ldots, m+1\}, \text{ otherwise.} \end{cases}$$

Denote by $L_i = \bigcap\limits_{k \in \{2,\ldots,q\}} L_i(k)$. Following the previous remarks, whenever $3 \le |L_i(k)| \le d$ for some k the problem has no solution. Moreover, if $|L_i| = m+1$, then no replacements are necessary in the arrays $M_i, \ldots, M_{i+d-1}$. Thus, the only cases we should analyse are $|L_i| \in \{0, 1, 2\}$.

Example 1. *Consider $d = 4$ and the arrays*

$$\begin{array}{ll} M_1 = (a \quad a \quad f \quad a) & M^b_1 = (1 \quad 1 \quad 0 \quad 1) \\ M_2 = (f \quad b \quad b \quad b) & M^b_2 = (1 \quad 0 \quad 0 \quad 0) \\ M_3 = (c \quad f \quad c \quad c) & M^b_3 = (1 \quad 0 \quad 1 \quad 1) \\ M_4 = (d \quad d \quad f \quad d) & M^b_4 = (1 \quad 1 \quad 0 \quad 1) \\ M_5 = (e \quad e \quad e \quad f) & M^b_5 = (1 \quad 1 \quad 1 \quad 0) \end{array}$$

Following Remark 5, we can apply only one Type 2 *replacement in the first 4 arrays, and only one* Type 2 *replacement in the last 4 arrays. However, we see that $M^b_1[3] = M^b_2[3] = M^b_4[3] = 0$, and thus, by Remark 6, we have that among the first, second and fourth arrays we need two* Type 2 *substitutions. This shows that there is no solution. Looking at the last 4 arrays, however, we see that $L_2 = \{2\}$. In this case, applying on M_2 a* Type 2 *substitution of f with $\diamond$, and on all other three arrays* Type 1 *replacements, satisfies our conditions (for these* 4 *columns only).*

We analyse each case assuming that $3 \le |L_i(k)| \le d$ is false, for all j.

Case 1. $|L_i| = 0$, thus, $L_i = \emptyset$.

If there exists a unique array M_j such that M^b_j contains zeros we get a contradiction with $L_i = \emptyset$. The case $q = 2$ leads immediately to a contradiction since now we have $L_i = L_i(2) \neq \emptyset$.

Assume $q \ge 3$. If there exist k with $1 \le k \le q$ and $j_1, j_2 \in \{i, \ldots, i+d-1\}$ with $j_1 \neq j_2$, such that $M^b_{j_1}[k] = M^b_{j_2}[k] = 0$, then we claim no replacement can be applied to these arrays. Indeed, following Remarks 5 and 6 we assume without loss of generality that we apply a *Type 2* replacement to M_{j_1} and a *Type 1* replacement to M_{j_2}. Since $L_i = \emptyset$, there exist $j_3 \in \{i, \ldots, i+d-1\}$

and $k' \neq k$ with $1 \leq k' \leq q$, such that $M^b_{j_1}[k'] = 1$ and $M^b_{j_3}[k'] = 0$. If we apply a *Type 1* replacement to M_{j_3}, then we have holes at position k' of both M_{j_1} and M_{j_3}. If we apply a *Type 2* replacement to M_{j_3}, then we have holes at position 1 of both M_{j_2} and M_{j_3}. A contradiction is reached in both cases.

Now assume that there exist no k with $1 \leq k \leq q$ and $j_1, j_2 \in \{i, \ldots, i+d-1\}$ with $j_1 \neq j_2$, such that $M^b_{j_1}[k] = M^b_{j_2}[k] = 0$. If there are k_1 and k_2 with $1 \leq k_1, k_2 \leq q$ and $k_1 \neq k_2$, and j_1 and j_2 with $j_1, j_2 \in \{i, \ldots, i+d-1\}$ and $j_1 \neq j_2$, such that $M^b_{j_1}[k_1] = M^b_{j_2}[k_2] = 0$, then $M^b_{j_1}[k_2] = M^b_{j_2}[k_1] = 1$. It follows that we cannot apply *Type* 2 replacements to any of these arrays, but *Type* 1 replacements must be applied to both of them.

Therefore, if $L_i = \emptyset$ the problem has a solution only when no two arrays contain 0 at the same position (i.e., $|L_i(k)| \leq 1$ for all k), and in this case we must apply *Type* 1 replacements to all the arrays that contain 0.

In conclusion, if $L_i = \emptyset$ and $|L_i(k)| \leq 1$ for all $k \in \{1, \ldots, q\}$ then $\phi_i = 1$.

Case 2. $|L_i| = 1$, let $L_i = \{j_0\}$.

In this case, the array $M^b_{j_0}$ has elements equal to 0 at every position where M^b_j has also 0, for $j \in \{i, \ldots, i+d-1\}$.

If $q = 2$, all the integers j such that M^b_j contains a 0 are in L_i. Thus, there exists an unique array $M^b_{j_0}$ that contains exactly one 0, and either a *Type 1* or a *Type 2* replacement can be applied to it.

Assume that $q \geq 3$. If there exists at least one other array M_{j_1}, such that $M^b_{j_1}$ does not contain only 1's, then it is mandatory to apply a *Type 2* replacement to M_{j_0}. Indeed, if $M^b_{j_0}[k] = 0$, then $M^b_{j_1}[k] = 0$ and there exists k', such that $M^b_{j_0}[k'] = 0$ and $M^b_{j_1}[k'] = 1$, since, otherwise, j_1 would be in L_i as well. It follows that we cannot apply *Type 1* replacement to M_{j_0}.

To conclude, if more than one array M^b_j contains 0, then we must apply a *Type 2* replacement to M_{j_0} and *Type 1* replacements to all the other arrays, if possible. If exactly one array contains 0 then either a *Type 1* or a *Type 2* replacement can be applied to it.

Formally, if $L_i = \{j_1\}$ and if at least two of the arrays $M^b_i, M^b_{i+1}, \ldots, M^b_{i+d-1}$ contain 0 then $\phi_i = x_{j_1}$; otherwise, $\phi_i = 1$.

Case 3. $|L_i| = 2$, let $L_i = \{j_1, j_2\}$.

In this case, it is easy to see that $M^b_{j_1} = M^b_{j_2}$. Using arguments similar to above, we show that when other arrays contain 0's, the problem has no solution, or, otherwise, a *Type 2* replacement must be applied to exactly one of the arrays M_{j_1} or M_{j_2} and *Type 1* replacements to the other array.

Formally, if $L_i = \{j_1, j_2\}$ then $\phi_i = x_{j_1}$ **xor** x_{j_2}.

The above discussion shows that the set L_i indicates the way *Type 2* replacements should be applied to the arrays $M_i, \ldots, M_{i+d-1}$ in order to make them satisfy both the periodicity and the d-validity condition.

Assume now that the arrays $M_i, \ldots, M_{i+d-1}$ do not have the same length and set s to be the minimum number of elements of an array. We define the arrays M_j^∇ formed of only the first s elements of M_j, where $j \in \{i, \ldots, i+d-1\}$. In this case, we either obtain a formula ϕ_i^∇ for the arrays $M_i^\nabla, \ldots, M_{i+d-1}^\nabla$, just as above, or conclude that the problem has no solution. If a formula is computed, then we have that $\phi_i^\nabla = x_j$; or, $\phi_i^\nabla = x_{j_1}$ **xor** x_{j_2}; or, $\phi_i^\nabla = 1$.

When $\phi_i^\nabla = x_j$ we have that:

- there exist $j_1, j_2 \in \{i, \ldots, i+d-1\} \setminus \{j\}$ and $s' > s$, such that $M_{j_1}^b[s'] = M_{j_2}^b[s'] = 0$, and the problem has no solution.
- there exist $j_1 \in \{i, \ldots, i+d-1\} \setminus \{j\}$ and $s' > s$, such that $M_{j_1}^b[s'] = 0$ and $M_j^b[s'] = 1$, and the problem has no solution.
- otherwise, we set $\phi_i = x_j$.

In the second case, when $\phi_i^\nabla = x_{j_1}$ **xor** x_{j_2}, we have that:

- there exist $j_1', j_2' \in \{i, \ldots, i+d-1\} \setminus \{j_1, j_2\}$ and $s' > s$, such that $M_{j_1'}^b[s'] = M_{j_2'}^b[s'] = 0$, and the problem has no solution.
- there exist $j_1' \in \{i, \ldots, i+d-1\} \setminus \{j_1, j_2\}$ and $s' > s$, such that $M_{j_1'}^b[s'] = 0$ and $M_{j_1}^b[s'] = M_{j_2}^b[s'] = 1$, and the problem has no solution.
- there exist $j_1' \in \{i, \ldots, i+d-1\} \setminus \{j_1, j_2\}$ and $s' > s$, such that $M_{j_1'}^b[s'] = M_{j_1}^b[s'] = 0$ and $M_{j_2}^b[s'] = 1$, and we set $\phi_i = x_{j_1}$.
- there exist $j_1' \in \{i, \ldots, i+d-1\} \setminus \{j_1, j_2\}$ and $s' > s$, such that $M_{j_1'}^b[s'] = 0$, $M_{j_1}^b[s'] = 1$ and $M_{j_2}^b[s'] = 0$, and we set $\phi_i = x_{j_2}$.
- otherwise, we set $\phi_i = x_{j_1}$ **xor** x_{j_2}.

Finally, when $\phi_i^\nabla = 1$, we have that:

- there exist 0's in more than one of the truncated arrays and it is mandatory to apply only *Type* 1 replacements to these arrays. If there exist an $s' > s$ and $j_1, j_2 \in \{i, \ldots, i+d-1\}$ such that $M_{j_1}^b[s'] = M_{j_2}^b[s']$ we set $\phi_i' = 0$, and, otherwise, we leave $\phi_i' = 1$.
- exactly one of the truncated arrays, say M_j^b, contains 0's.

– If there exist $j_1, j_2 \in \{i, \ldots, i+d-1\} \setminus \{j\}$ and $s' > s$, such that $M_{j_1}^b[s'] = M_{j_2}^b[s'] = 0$, then the problem has no solution.

– If the above case does not hold and there exist $j_1 \in \{i, \dots, i+d-1\} \setminus \{j\}$ and $s' > s$, such that $M^b_{j_1}[s'] = 0$ and $M^b_j[s'] = 0$, then we must apply a *Type 2* replacement to M_j and set $\phi_i = x_j$.

– Otherwise, we set $\phi_i = 1$.

- no 0's are in the truncated arrays and we let s' be the minimum number such that there exists $j \in \{i, \dots, i+d-1\}$ with $M^b_j[s'] = 0$. We keep only the first s' symbols of any array M_j with $j \in \{i, \dots, i+d-1\}$ that has at least s' symbols and try to obtain a formula for them. The arrays M^b_j with $j \in \{i, \dots, i+d-1\}$ are truncated accordingly. Clearly, the only 0's that appear in the newly truncated arrays M^b_j with $j \in \{i, \dots, i_d - 1\}$ can only be placed at the last positions in these arrays. If there exist more than one array containing 0's, then the formula we obtain for them is different from 1. Next, we repeat the analysis described above, for the original arrays.

Algorithm 4 summarises all the facts explained above. Assume that in each formula $\phi_i = x_{j_1}$ **xor** x_{j_2} we have $j_1 < j_2$. In this setting, x_{j_1} is said to be the first variable of the formula, while x_{j_2} is the second one. If the formulas consist of a single variable $\phi_i = x_{j_1}$, then this variable is said to be both the first and the second one, for uniformity.

Algorithm 4 *InitialFormula*(M, i): returns ϕ_i.

```
1: If $M_i, \dots, M_{i+d-1}$ have different lengths, truncate them, as described.
2: Compute $L_i(k)$ as stated before. If a set $L_i(k)$ has $\ell$ elements, $3 \le \ell \le d$,
   set $\phi_i = 0$. Otherwise, set $L_i = \bigcap_{k \in \{2,\dots,q\}} L_i(k)$.
3: if $L_i = \{j_1, j_2\}$ then
4:     $\phi_i = x_{j_1}$ xor $x_{j_2}$
5: elseif $L_i = \{j_1\}$ then
6:     if at least two of the arrays $M^b_i, M^b_{i+1}, \dots, M^b_{i+d-1}$ contain 0 then
7:         $\phi_i = x_{j_1}$
8:     else $\phi_i = 1$
9: elseif $|L_i| = m + 1$ or $|L_i(k)| \le 1$, for all $k \in \{2, \dots, q\}$ then
10:     $\phi_i = 1$
11: else $\phi_i = 0$
12: If $M_i, \dots, M_{i+d-1}$ were truncated, then update $\phi_i$, if necessary, to ensure
    the $d$-validity condition. The update is done as previously explained.
13: Return $\phi_i$.
```

Using similar arguments we can easily obtain the following remarks:

Remark 8. *Consider a variable x_i and the formulas ϕ_j for $1 \leq j \leq m-d+1$.*
– If x_i does not appear in any formula ϕ_j, then in all solutions of the problem we can have a Type 1 *replacement to array i (a* Type 2 *replacement to such an array can be omitted).*
– If x_i appears in one of the formulas, then there exist ℓ with $0 < |\ell - i| < d$ and a position k, such that $M_i^b[k] = 0 = M_\ell^b[k]$. Thus, $M_i[k]$ cannot be replaced with a hole in array i and all symbols different from $M_i[k]$ are replaced when a Type 2 *replacement is performed to this array.*
*– If $\phi_i = x_j$, or $\phi_i = x_{j'}$ **xor** x_j with $j' < j$, then the formulas $\phi_{i+1}, \ldots, \phi_{j+d}$ can only have one of the following forms: $x_{j'}$ **xor** x_j, x_j, 1, 0, x_j **xor** x_k, or x_k, for some $k > j$ and $|k-i| \geq d$. Once a new variable x_k appears in one of these formulas, the first two mentioned possibilities cannot occur any more.*

Algorithm 5 *Simplify*$(\phi_1, \ldots, \phi_{m-d+1})$: returns $F_1, \ldots, F_m$, sets of formulas.

```
 1: Let $F = \{\phi_1, \ldots, \phi_{m-d+1}\}$.
 2: for $i \in \{1, \ldots, m\}$ do
 3:    Let $F'_i$ be the set of all formulas from $F$ with $x_i$ as second variable.
 4:    if $x_i$ is a formula in $F'_i$ then
 5:        Set $F'_i = \{x_i\}$
 6:    else $F'_i$ can only contain exactly one formula $x_j$ xor $x_i$, or it is empty.
 7: end for
 8: Let $F = \bigcup_{i \in \{1,\ldots,m\}} F'_i$.
 9: for $i \in \{1, \ldots, m\}$ do
10:    Let $F_i$ be the set of all formulas from $F$ with $x_i$ as first variable.
11:    if $x_i$ is a formula in $F_i$ then
12:        Set $F_i = \{x_i\}$
13:    else $F_i$ can only contain exactly one formula $x_i$ xor $x_j$, or it is empty.
14: end for
15: Return the sets $F_1, \ldots, F_m$.
```

It only remains to show how we can construct ϕ_M. It is obvious that, to transform M in a periodic d-valid sequence we must transform each subsequence of d consecutive arrays in M in a periodic d-valid sequence. Thus, all formulas ϕ_i are satisfied, or, when any equals 0, the problem has no solution. Let us assume that all the formulas are different from 0. However, ϕ_M cannot just be the conjunction of all these formulas, since we may get in a situation in which two variables x_{j_1} and x_{j_2}, appearing in different formulas ϕ_{i_1} and ϕ_{i_2}, respectively, for which $|j_1 - j_2| \leq d-1$, are both set to 1. Furthermore, we should check if we can apply a *Type 2* replacement to an array M_i and *Type* 1 replacements to the $d-1$ arrays that precede and $d-1$ arrays that succeed it.

For this, we firstly run $Simplify(\phi_1, \ldots, \phi_{m-d+1})$, described in Algorithm 5, to ensure that each variable is the first, respectively last, in at most one formula.

Second, we simplify the formulas even more by eliminating the variables that cannot be true. In particular, we detect the arrays in which we should apply a *Type 2* replacement and cannot because of the *Type 1* replacements on the $d-1$ arrays that precede and the $d-1$ arrays that succeed it. This is done by Algorithm 6, $Filter(F_1, \ldots, F_M)$, where $F_1, \ldots, F_M$ represent sets of formulas whose first variables are $x_1, \ldots, x_m$. Note that, each F_i contains at most one formula. Moreover, this function computes a series of equalities $x_i = x_i^s$ for some $i \in \{1, \ldots, m\}$ and $s \in V$ that indicate which symbol should not be transformed into a $\diamond$ when a *Type 2* replacement is applied to M_i. This information is needed in order to effectively produce a solution for the problem. Finally, remark that Algorithm 6 also ensures that we cannot perform a *Type* 2 replacement to an array to which a *Type* 1 replacement was necessary, according to the analysis made in *Case 1* previously discussed.

In the end, starting from the simplified list of formulas we can construct ϕ_M by running $Formula(\phi_1, \ldots, \phi_{m-d+1})$, described in Algorithm 7. In fact this algorithm detects some redundant restrictions that appear in the list of formulas and eliminates them. During this processing we also identify cases when the problem has no solution. More precisely, we first obtain the sets of formulas $F_1, \ldots, F_m$ and the set of equalities U by running consecutively $Filter(Simplify(\phi_1, \ldots, \phi_{m-d+1}))$. Then, we modify iteratively these sets accordingly. For two formulas $\rho = x_{j_1}$ **xor** x_{j_2} and $\phi = x_{j_1'}$ with $j_1' - j_1 \geq d$ and $j_1' - j_2 \leq d-1$, we delete ρ and add the atomic formula x_{j_1}. That is, on the array j_1' we must apply a Type 2 replacement, but we also must apply such a replacement on one of the arrays j_1 or j_2. Since a *Type 2* replacement cannot be applied simultaneously on j_1' and j_2 we can only apply such replacements on j_1 and j_1'. Following similar reasons, whenever we have two formulas $\rho = x_{j_1}$ and $\phi = x_{j_1'}$ **xor** $x_{j_2'}$ with $j_2' - j_1 \geq d$ and $j_1' - j_1 \leq d-1$, we delete ϕ and add the atomic formula $x_{j_2'}$. Finally, for two formulas $\rho = x_{j_1}$ and $\phi = x_{j_1'}$ with $j_1' - j_1 \leq d-1$ we have no solution, since two Type 2 replacements should

Algorithm 6 *Filter*$(F_1, \dots, F_m)$: returns $F_1, \dots, F_m$, and a set of equalities U.

```
1: Let $U$ be an empty set and let $\{x_i^s \mid s \in V, i \in \{1, \dots, m\}\}$ be a set of
   new variables.
2: for $i \in \{1, \dots, m\}$ do
3:    if $F_i = \{x_i\}$ then
4:       ($A$) check if there exists a unique symbol $s \neq M_i[1]$ in $M_i$ that
         cannot be replaced by $\diamond$ when a Type 2 replacement is applied to
         $M_i$, and Type 1 replacements are applied to the $d-1$ arrays that
         precede and the $d-1$ arrays that succeed $M_i$ in $M$, without violating
         the $d$-validity condition.
5:        if check ($A$) is negative, then set $F_i = \{0\}$.
6:    elseif $F_i = \{x_i$ xor $x_j\}$ then
7:       ($B$) check if there exists a unique symbol $s \neq M_i[1]$ in $M_i$ that
         cannot be replaced by $\diamond$ when a Type 2 replacement is applied to
         $M_i$, and Type 1 replacements are applied to the $d-1$ arrays that
         precede and the $d-1$ arrays that succeed $M_i$ in $M$, without violating
         the $d$-validity condition.
8:       ($C$) check if there exists a unique symbol $s' \neq M_j[1]$ in $M_j$ that
         cannot be replaced by $\diamond$ when a Type 2 replacement is applied to $M_j$,
         and Type 1 replacements are applied to the $d-1$ arrays that precede
         and the $d-1$ arrays that succeed $M_j$ in $M$, without violating the
         $d$-validity condition.
9:        if check ($B$) and check ($C$) are positive, then
10:          $F_i = \{x_i$ xor $x_j\}$ and add equalities $x_i = x_i^s$ and $x_j = x_j^{s'}$ to $U$.
11:       elseif check ($B$) is positive, then
12:          $F_i = \{x_i\}$ and add the equality $x_i = x_i^s$ to $U$.
13:       elseif check ($C$) is positive, then
14:          $F_i = \emptyset$ and $F_j = \{x_j\}$.
15: end for
16: Return the sets $F_1, \dots, F_m$ and the set $U$.
```

be applied in d consecutive lines, and, thus, we set $\phi_M = 0$. Once all the possible such modifications are applied, we get ϕ_M as the conjunction of all these formulas.

The soundness of the three Algorithms is straightforward, since we just modify the list of formulas to eliminate the situations mentioned above.

According to the previous remarks, it is not difficult to see that Problem 3

Algorithm 7 *Formula*$(\phi_1, \ldots, \phi_{m-d+1})$: returns ϕ_M and a set of equalities U.

```
1: If any of the input formulas equals 0, **Return** $\phi_M = 0$. Otherwise, eliminate
   all the input formulas equal to 1 and if nothing remains, **Return** $\phi_M = 1$.
2: Let $F'_1, \ldots, F'_m = Simplify(\phi_1, \ldots, \phi_{m-d+1})$.
3: Let $F_1, \ldots, F_m, U = Filter(F'_1, \ldots, F'_m)$.
4: Let $i_0 = \min\{i \mid i \in \{1, \ldots, m\}, F_i \neq \emptyset\}$ and assume that $F_{i_0} = \{\rho\}$.
5: Let $S$ be an empty stack. $S.push(\rho)$. Let $i = 1$.
6: **while** $i \leq m$ **do**
7:    Assume that $F_i = \{\phi\}$ and the formula on top of the stack is $\rho$.
8:    **if** $\rho = x_{j_1}$ **xor** $x_{j_2}$ and $\phi = x_{j'_1}$ **xor** $x_{j'_2}$ **then**
          {We have two cases (according to Remark 8):}
9:        1. $j'_2 - j_1 \geq d$, $j'_1 = j_2$: $S.push(\phi)$, $i++$, and **go to 6**.
10:       2. $j'_1 - j_2 \geq d$: $S.push(\phi)$, $i++$, and **go to 6**.
11:   **if** $\rho = x_{j_1}$ **xor** $x_{j_2}$ and $\phi = x_{j'_1}$ **then**
          {We have two cases (according to Remark 8):}
12:       1. $j'_1 - j_1 \geq d$, $j'_1 - j_2 \leq d-1$: $S.pop(\rho)$, $F_{j_1} = \{x_{j_1}\}$, $i = j_1$, and **go to 6**.
13:       2. $j'_1 - j_2 \geq d$: $S.push(\phi)$, $i++$, and **go to 6**.
14:   **if** $\rho = x_{j_1}$ and $\phi = x_{j'_1}$ **xor** $x_{j'_2}$ **then**
          {We have two cases (according to Remark 8):}
15:       1. $j'_2 - j_1 \geq d$, $j'_1 - j_1 \leq d - 1$: Let $F_{j'_2} = \{x_{j'_2}\}$, $i++$, and **go to 6**.
16:       2. $j'_2 - j_1 \geq d$, $j'_1 - j_1 \geq d$: $S.push(\phi)$, $i++$, and **go to 6**.
17:   **if** $\rho = x_{j_1}$ and $\phi = x_{j'_1}$ **then**
          {We have two cases (according to Remark 8):}
18:       1. $j'_1 - j_1 \leq d - 1$: **Return** $\phi_M = 0$ and exit the algorithm.
19:       2. $j'_1 - j_1 \geq d$: $S.push(\phi)$, $i++$, and **go to 6**.
20: **end while**
21: Let $\phi_M$ be the conjunction of all the formulas of $S$.
22: **Return** $\phi_M$ and the set of equalities $U$.
```

has a positive solution for the input sequence of arrays M if and only if there exists a truth assignment for the variables $x_1, \ldots, x_m$ that makes ϕ_M equal to 1. Clearly, instead of each formula x_i **xor** x_j that appears in ϕ_M we can write the formula $(x_i \vee x_j) \wedge (\overline{x_i} \vee \overline{x_j})$. Hence, ϕ_M is a formula in 2-Conjunctive Normal Form. We can solve this efficiently using a linear time solution of the 2-CNF-SAT Problem, see [23].

However, a more careful analysis shows that, the special form of the formula ϕ_M, as it results from Algorithm 7, allows us to decide whether it is satisfiable or not far easier, since ϕ_M is not decidable only when it equals 0.

Otherwise, to find a truth assignment that makes $\phi_M = 1$, we assign $x_i = 1$ for all clauses (x_i) present in ϕ_M, delete these variables from ϕ_M, and for the rest of the variables that appear in the formula we choose an assignment that makes all the clauses true. The latter part can be done quite easily:

- if $(x_i \textbf{ xor } x_j)$ is now the first clause of ϕ_M, then set $x_i = b$ and $x_j = \overline{b}$, for $b \in \{0, 1\}$.
- as we have seen, x_j can only appear in the second clause.
- if the second clause is $x_j \textbf{ xor } x_t$, then set $x_t = \overline{x_j}$, and try to find an assignment satisfying the rest of the clauses in a similar manner.
- if the second clause is $x_{j'} \textbf{ xor } x_t$ with $j', t > j$, then set $x_{j'} = b$ and $x_t = \overline{b}$ for $b \in \{0, 1\}$ and try to find an assignment satisfying the rest of the clauses in a similar manner.

Algorithm 8 *Solve*(M, d): returns **true** if Problem 3 has a solution.

```
1: for $i \in \{1, \ldots, m - d + 1\}$ do
2:     Let $\phi_i = InitialFormula(M, i)$.
3: end for
4: Let $\phi_M, U = Formula(\phi_1, \ldots, \phi_{m-d+1})$.
5: if there exists an assignment for $x_1, \ldots, x_m$ such that $\phi_M = 1$ then
6:     Return true
       {For a solution we apply Type 2 replacements to all $M_i$ with $x_i = 1$
       (the symbols $s$ not replaced by $\diamond$'s are those for which the equality
       $x_i = x_i^s$ is in $U$), and Type 1 replacements to all the other arrays.}
7: else Return false.
```

Thus, a solution for Problem 3 is given by Algorithm 8.

4.2. Complexity aspects

The main result for this section is the following:

Theorem 3. *Problem 3 can be decided in linear time. An instance for which the problem answers positively is obtained in the same time complexity using Algorithm 8.*

The fact that Algorithm 8 solves Problem 3 follows from the previous discussions. However, in order to show the complexity part we discuss several implementation details of the aforementioned algorithms.

First, recall that the number of memory cells needed to store the input, i.e., the size of the input, is $\mathcal{O}(mq)$.

We show that the formula ϕ_M is constructed in linear time and, since the formula has $\mathcal{O}(mq)$ clauses and variables, deciding its satisfiability and finding an assignment for the variables that makes $\phi_M = 1$ takes also linear time. Hence, deciding if Problem 3 has a solution and effectively finding one is done in $\mathcal{O}(mq)$ time. The proof relies on several observations that ensure that all previously described procedures are implemented in $\mathcal{O}(mq)$ time.

Note that the sets used in the algorithms of the previous Section are always implemented as queues. Moreover, the sequence M^b is constructed in time $\mathcal{O}(mq)$, canonically.

Let us now investigate the time complexity for Algorithm 4. In particular, we show how we compute efficiently the sets L_i and formulas ϕ_i, for all i:

- fix $i \in \{1, \dots, m-d+1\}$ and define for each $k \in \{1, \dots, q\}$ an ordered queue Q_k that contains, when L_i is computed, the integers $\{i_1, \dots, i_t\}$ such that $M^b_{i_j}[k] = 0$ and $0 \leq i_j - i < d$, for all $j \in \{1, \dots, t\}$. Moreover, define for each $k \in \{1, \dots, q\}$ a variable p_k that stores the cardinality of Q_k.

- if any $p_k \geq 3$, when L_i is computed, we decide that $\phi_i = 0$. Otherwise, we determine in $\mathcal{O}(q)$ time the elements that appear in all the non-empty queues Q_k (each has at most 2 elements), and return these as the set L_i. Moreover, we compute in $\mathcal{O}(q)$ time the formula ϕ_i, as described in Algorithm 4.

- note that, when $i = 1$, we construct Q_k for $k \in \{1, \dots, q\}$ by simply traversing the arrays $M_1, \dots, M_d$, which takes $\mathcal{O}(dq)$ time. Then, to update the queues, when we move from computing L_i and the formula ϕ_i, to computing L_{i+1} and the formula ϕ_{i+1}, we just delete the minimum element from all these queues, if it equals i, and add to the queue Q_k the element $i + d$, for all k's such that $M^b_{i+d}[k] = 0$. These operations take, again, $\mathcal{O}(q)$ time.

- following the above, the time needed to compute all formulas ϕ_i for $i \in \{1, \dots, m-d+1\}$ is $\mathcal{O}(mq)$.

It is not hard to see that Algorithm 5 runs in $\mathcal{O}(m)$ time.

Let us now analyse the time complexity of Algorithm 6:

- the verifications from its iterative block are performed efficiently using a strategy similar to the one described for Algorithm 4.

- moreover, recall that a symbol can be replaced by a hole in array M_i using a *Type* 2 replacement if and only if none of the $d-1$ arrays that precede it, nor the $d-1$ arrays that succeed it have 0 at that position. Thus, we define for each $k \in \{1, \ldots, q\}$ two ordered queues Q'_k and Q''_k, that contain, at the moment when F_i is checked, the integers $\{i_1, \ldots, i_t\}$, such that $M^b_{i_j}[k] = 0$ and $0 < i_j - i < d$, and, respectively, the integers $\{i'_1, \ldots, i'_s\}$, such that $M^b_{i'_\ell}[k] = 0$ and $0 < i - i'_\ell < d$, for all $j \in \{1, \ldots, t\}$ and $\ell \in \{1, \ldots, s\}$. These queues are computed and updated just as in the previous case and none contains more than two elements at a given moment. For our verifications, it suffices to check if the set $\{M_i[j] \mid \text{with } Q'_j \text{ or } Q''_j \text{ not empty}\}$ contains exactly one symbol. If so, the verification returns a positive answer. Otherwise, it returns a negative one. Clearly, the time needed to do this is $\mathcal{O}(mq)$.
- consequently, the total running time needed by Algorithm 6 is $\mathcal{O}(mq)$.

It is not hard to see that Algorithm 7 runs in $\mathcal{O}(m)$ time. Moreover, if, at any point, a formula ϕ on top of the stack S, has x_j as the second variable, then every variable x_i that appears in a formula from S fulfils $i < j$.

Summarising the above, it follows that Algorithm 8 works in $\mathcal{O}(mq)$ time.

4.3. The main problem

In the following, Problem 2 is solved in $\mathcal{O}(|w|)$ time using the previously discussed solution of Problem 3, where the constant hidden by the $\mathcal{O}$-denotation does not depend on either d or p.

Let us first analyse the case when $d \leq p$. Define the sequence of $p+d-1$ arrays M. For $i \leq p$, the array M_i contains all symbols $w[x]$ with $x \geq 1$ and $x \equiv i \textbf{ mod } p$. For j such that $1 \leq j \leq d-1$ the array M_{p+j} contains the symbols $w[p+x]$ with $x \geq 1$ and $x \equiv j \textbf{ mod } p$. Finally, construct the formula ϕ_M as previously explained. Note that in the case when $d = 1$ the formula is canonically equal to 1.

Each array M_i is associated with a variable x_i, as previously explained, for all $i \in \{1, \ldots, p+d-1\}$. However, when $j \equiv i \textbf{ mod } p$, the variables x_i and x_j are not independent, since the symbols that appear in these two columns correspond to a single symbol appearing on a certain position in the initial word. Thus, we cannot replace a certain symbol with $\diamond$ in any of the lines and leave it unchanged in the other. To capture this dependency in our final formula, we introduce additional variables that appear in the set of equalities U, produced by the function *Filter*. First we obtain the set U' in the following manner:

- if $x_i = x_i^s$ is in U, then we put in U' the formula $(x_i \vee \overline{x_j^s}) \wedge (\overline{x_i} \vee x_j^s)$, provided that $j \in \{1, \ldots, p\}$ and $i \equiv j(\textbf{ mod } p)$.
- if $x_i = x_i^s$ is in U for some integer $i > p$, then we put in U' the formula $\overline{x_i} \vee x_j^s$, for $j \in \{1, \ldots, p\}$ such that $i \equiv j(\textbf{ mod } p)$ and $s = M_i[1]$.

Next, we make the conjunction between ϕ_M and all the formulas in U', and replace each formula z **xor** y that appears in ϕ_M with $(z \vee y) \wedge (\overline{z} \vee \overline{y})$, in order to obtain a new formula ϕ'_M in 2-Conjunctive Normal Form.

Problem 2 has a solution if and only if there exists a truth assignment for the variables $\{x_1, \ldots, x_{p+d-1}\} \cup \{x_i^s \mid s \in V \text{ and } i \in \{1, \ldots, p\}\}$ that gives $\phi'_M = 1$. A solution is constructed applying *Type 2* replacements to all M_i with $x_i = 1$ and *Type 1* replacements to all other arrays, followed by corresponding replacements in the input word. For the *Type 2* replacements, the symbols not replaced by $\diamond$'s are the symbols s, where the equality $x_i = x_i^s$ is in U.

Clearly, if $d \leq p + 1$, then it is quite easy to implement all of the previous steps in linear time with respect to the length of the input word w. The time needed to solve the problem does not depend on either d or p.

The case when $d > p$ is very simple. First we define the sequence of arrays M, such that for $i \leq p$ the array M_i contains all the symbols $w[x]$ with $x \geq 1$ and $x \equiv i \textbf{ mod } p$.

Some of the symbols of the arrays must be replaced in order to ensure that for every M_i there exists a symbol s_i that contains all other symbols of the array.

Let M_{i_0} be the first array that contains at least two different symbols. Although, several symbols of this array must be replaced with holes, note that it is not the case for any two consecutive symbols, since this violates the d-validity condition. It follows that there are at most two possibilities to choose the symbol that is not replaced by $\diamond$ in the array. Assume now that we choose to keep a symbol a in this array and we replace all other symbols with $\diamond$'s, such that the aforementioned condition is fulfilled. This replacement determines on each array M_i of M a symbol a_i that is kept unchanged. Thus, we already determined all the symbols of M that should be replaced with $\diamond$.

We only have to check if the word that we obtain is d-valid. If yes, we have a solution for the problem. Otherwise, we try the other choice for the symbol a in the M_i's. Finally, if by using this strategy, we cannot find a valid replacement, the problem has no solution.

It is not hard to see that the time needed to solve the problem in this case is also $\mathcal{O}(|w|)$ and does not depend on either d or p. Clearly, this time is optimal. In conclusion, we state the following Theorem.

Theorem 4. *Problem 2 can be decided in linear time. A solution for this problem is obtained in the same time complexity.*

This theorem improves the result in [15], where Problem 2 was solved in time $\mathcal{O}(nd)$. It is worth noting that our approach is completely different from the one in the cited paper.

4.4. Looking for the optimal number of holes

In this final section we approach some optimisation related to Problem 2:

Problem 4. *Given a word $w \in V^*$ and two positive integers d and p, both less than $|w|$, construct a p-periodic d-valid partial word contained in w, and having a minimum number of holes.*

The solution of this problem is based on the solution of Problem 2 presented in the previous section. Let us assume, without losing generality, that Problem 2 has a solution for the input word w and numbers d and p. As a first step we construct for the word w and integer p, the sequence of arrays M using Algorithm 7, formula ϕ_M and set of equalities U.

In addition to Remark 8, note that, if a variable x_i does not appear in ϕ_M and a *Type* 2 replacement can be applied to the array i in a correct solution of Problem 2, then both the $d-1$ arrays that precede the array M_i^b and the $d-1$ arrays that succeed it contain only 1's. Although for a solution of Problem 2 it is not really important whether we apply a *Type* 1 or a *Type* 2 replacement to this array, in the case of the associated optimisation Problem 4, when only a minimal number of holes is wanted, it is important to use on array i a replacement that leads to such a solution. Therefore, whenever the symbol that occurs the most times in array i is s and this is not the first symbol of the array, we update ϕ_M by making the conjunction with x_i, and add to U the equality $x_i = x_i^s$. Clearly, in the formula ϕ_M there exists no variable x_j, such that $|j-i| < d$.

For simplicity assume that all variables x_i with $i \in \{1, \ldots, p+d-1\}$ not appearing in ϕ_M are set to 0.

We can associate with each variable x_i two values, denoted $v[i][1]$ and $v[i][2]$ for $i \in \{1, \ldots, p\}$ such that $v[i][1]$ equals the number of holes that are inserted in the array M_i if we set $x_i = 1$, and $v[i][2]$ equals the number of holes that are inserted in the array M_i if we set $x_i = 0$. All these can be computed in linear time, using the set U. Clearly, the solution of Problem 4 is given by a truth assignment of the variables $x_1, \ldots, x_p, x_{p+1}, \ldots, x_{p+d-1}$ such that it makes the formula ϕ_M true and minimises

$$\sum_{i \in \{1,\ldots,p\}} (x_i v[i][1] + (1 - x_i) v[i][2]).$$

The truth value of the variable x_{p+j} associated with the array M_{p+j} where $1 \leq j \leq d-1$ is not independent from the truth value of the variable x_j, as already explained. For the first d arrays in the sequence M, ϕ_M contains a unique formula $\phi_1 = 1$, $\phi_1 = x_i$, or $\phi_1 = x_{i_1}$ **xor** x_{i_2}, such that $i, i_1, i_2 \in \{1, \ldots, d\}$. This formula is obtained by Algorithm 4 or was inserted in the above mentioned step (we identified an array i to which both types of replacements can be applied and chose a *Type* 2 replacement, since this led to a lower number of holes). In each case we get from ϕ_M two new formulas:

- If $\phi_1 = 1$, then we apply to all arrays $M_1, \ldots, M_d$ only *Type* 1 replacements. Thus, we can easily determine the truth values of the variables $x_{p+1}, \ldots, x_{p+d-1}$. We rewrite the formula ϕ_M according to the values that we have determined and have now a new formula ϕ'_M in which only variables x_j with $j \leq p$ appear. We also set $\phi''_M = 0$.

- If $\phi_1 = x_i$, then we perform *Type* 1 replacements to all first d arrays except for M_i, where we apply a *Type* 2 replacement. Again, we easily determine, using the set of equalities U, the truth values for the variables $x_{p+1}, \ldots, x_{p+d-1}$. Finally, we rewrite the formula ϕ_M according to the values that we have determined and have now a new formula ϕ'_M in which only variables x_j with $j \leq p$ appear. As before, we set $\phi''_M = 0$.

- If $\phi_1 = x_{i_1}$ **xor** x_{i_2}, then we have two new cases, according to the variable which is set to 1. However, using the set of equalities U, we can determine in both cases the truth values of the variables $x_{p+1}, \ldots, x_{p+d-1}$. Since setting one of the variables x_{i_1} or x_{i_2} to 1 may determine the truth values of some other variables x_j with $j \leq p$ that appear in ϕ_M, we can rewrite the formula ϕ_M in two ways. We obtain two new formulas ϕ'_M and ϕ''_M, respectively, in which only variables x_j with $j \leq p$ appear.

To solve Problem 4 we proceed as follows. First, we search for a truth assignment of the remaining variables that makes both ϕ'_M true and the sum

$\sum_{i\in\{1,\ldots,p\}}(x_i v[i][1]+(1-x_i)v[i][2])$ minimum among all the sums determined by other assignments for which $\phi'_M = 1$. Next, we search for an assignment that makes ϕ''_M true and the sum minimum, compared to the other assignments making ϕ''_M true. The answer to our problem is provided by one of these assignments, namely the one that yields the smallest sum.

Let ρ denote one of the formulas ϕ'_M or ϕ''_M. We already have an assignment of the variables $\{x_1,\ldots,x_j\}$ for some $j \geq d$ and the formula ρ contains only variables from $\{x_{j+1},\ldots,x_p\}$. Moreover, in an assignment that satisfies ϕ_M we must set $x_i = 1$ for all clauses (x_i) that appear in ρ. Let us assume that ρ is the conjunction of t clauses, each of these clauses being of the form x_{i_1} **xor** x_{i_2}, where $i_1 < i_2$. Let V denote a subset of $\{1,\ldots,p\}$ for which the value of x_i was already determined and set $S = \sum_{i\in V}(x_i v[i][1] + (1-x_i)v[i][2])$. Assume, also, that the clauses of ρ are ordered according to the index of their first variable and let $c(i)$ denote the i^{th} clause of ρ with respect to this order. Furthermore, denote by $f(i)$ and $s(i)$ the arrays to which the first and, respectively, the second variable of $c(i)$ are associated. For example, if $c(i) = x_{i_1}$ **xor** x_{i_2} with $i_1 < i_2$, then $f(i) = i_1$ and $s(i) = i_2$, while if $c(i) = x_{i_1}$, then $f(i) = s(i) = i_1$. Of course $f(i) > f(i-1)$, for all $i \in \{2,\ldots,t\}$.

According to the previously made remarks, the truth value of any variable that appears in $c(i)$ depends only on the values of the variables that appear in $c(i-1)$. This suggests that we can find in linear time an assignment of all the variables, that minimises the sum $\sum_{j\in\{1,\ldots,p\}\setminus V}(x_j v[j][1]+(1-x_j)v[j][2])$, by a dynamic programming strategy. We proceed as follows.

Let T_1 and T_2 be two arrays with t elements. For $i \in \{1,\ldots,t\}$ with $c(i) = x_{i_1}$ **xor** x_{i_2} we define the elements of these arrays as follows:

- $T_1[i] = S + \sum_{j\in\{1,\ldots,i_2\}\setminus V}(x_j v[j][1] + (1-x_j)v[j][2])$, where the truth values of the variables $x_1,\ldots,x_{i_1-1}$ were already determined such that $S + \sum_{j\in\{1,\ldots,i_1-1\}\setminus V}(x_j v[j][1] + (1-x_j)v[j][2])$ is minimum and these values permit the assignment $x_{i_1} = 1$ and $x_{i_2} = 0$. If such an assignment is impossible, then $T_1[i] = \infty$.

- $T_2[i] = S + \sum_{j\in\{1,\ldots,i_2\}\setminus V}(x_j v[i][1] + (1-x_j)v[i][2])$, where the truth values of the variables $x_1,\ldots,x_{i_1-1}$ were already determined such that $S + \sum_{j\in\{1,\ldots,i_1-1\}\setminus V}(x_j v[j][1] + (1-x_j)v[j][2])$ is minimum and these values permit the assignment $x_{i_1} = 0$ and $x_{i_2} = 1$. If such an assignment is impossible, then $T_2[i] = \infty$.

Whenever $c(i) = x_{i_1}$, we have $T_1[i] = T_2[i] = S + \sum_{j\in\{1,\ldots,i_2\}\setminus V}(x_j v[j][1] + (1 - x_j)v[j][2])$, where the truth values of the variables $x_1, \ldots, x_{i_1-1}$ were already determined such that $S + \sum_{j\in\{1,\ldots,i_1-1\}\setminus V}(x_j v[j][1] + (1 - x_j)v[j][2])$ is minimum.

It is rather simple noting that these arrays are computed in linear time. Clearly, the assignment that we are looking for is the one that makes the sum $\sum_{j\in\{1,\ldots,p\}\setminus V}(x_j v[j][1] + (1 - x_j)v[j][2])$ equal to $\min\{T_1[t], T_2[t]\}$. The values assigned to the variables are easily computed in linear time using dynamic programming, during the computation of the arrays T_1 and T_2.

We are now ready to state the main result of this section:

Theorem 5. *Problem 4 can be decided in linear time. A solution for this problem is obtained in the same time complexity.*

Once again this result is optimal, with respect to the time complexity.

It is interesting to note that in the case when $d = 1$ we can reformulate Problem 4 in a very simple way:

Problem 5. *Given a word $w \in V^*$ and positive integer $p < |w|$, construct a p-periodic partial word contained in w, having a minimum number of holes.*

The already presented solution can be particularised to this case giving us a new very simple and intuitive greedy solution for this problem. First, we construct for the word w and the integer p the sequence of arrays M, using Algorithm 7, and the formula ϕ_M. Naturally, we obtain $\phi_M = 1$. Next, we add to this formula all the variables x_i with $i \in \{1, \ldots, p\}$, such that applying a *Type* 2 replacement to the array i introduces a lower number of holes than applying a *Type* 1 replacement. Finally, we set all these variables to 1 and get a solution for the problem. In simpler words, in each of the arrays we replace with holes a minimum number of symbols such that the periodicity condition is fulfilled. The solution that we proposed for Problem 5 is also optimal with respect to the time complexity.

A final remark is that the same algorithmic strategy can be used to solve in linear time the following complementary optimisation problem: "Given a word $w \in V^*$ and two positive integers d and p, both less than $|w|$, construct a p-periodic d-valid partial word contained in w and having a maximum number of holes".

4.5. An Example

We end with a few examples regarding our algorithms.

First, we give a short example on how Problem 3 can be solved by our approach. Let $d = 4$, $m = 9$, $q = 3$, $k = 2$ and consider the sequence M and its corresponding sequence M^b:

$$\begin{array}{lcl} M_1 &=& (b \quad a \quad a) \\ M_2 &=& (b \quad b \quad b) \\ M_3 &=& (a \quad b \quad a) \\ M_4 &=& (d \quad d \quad d) \\ M_5 &=& (c \quad c \quad c) \\ M_6 &=& (c \quad e \quad c) \\ M_7 &=& (a \quad a) \\ M_8 &=& (b \quad b) \\ M_9 &=& (b \quad a) \end{array} \qquad \begin{array}{lcl} M^b_1 &=& (1 \quad 0 \quad 0) \\ M^b_2 &=& (1 \quad 1 \quad 1) \\ M^b_3 &=& (1 \quad 0 \quad 1) \\ M^b_4 &=& (1 \quad 1 \quad 1) \\ M^b_5 &=& (1 \quad 1 \quad 1) \\ M^b_6 &=& (1 \quad 0 \quad 1) \\ M^b_7 &=& (1 \quad 1) \\ M^b_8 &=& (1 \quad 1) \\ M^b_9 &=& (1 \quad 0) \end{array}$$

When we analyse the first $d = 4$ arrays, we obtain that $L_1 = \{1\}$. Since there is more than one array containing 0 we conclude that $\phi_1 = x_1$.

For the next d arrays we obtain that $L_2 = \{3\}$. Since there is exactly one array containing 0 we conclude that $\phi_2 = 1$.

The upcoming d arrays we look at are:

$$\begin{array}{l} M^b_3 = (1 \quad 0 \quad 1) \\ M^b_4 = (1 \quad 1 \quad 1) \\ M^b_5 = (1 \quad 1 \quad 1) \\ M^b_6 = (1 \quad 0 \quad 1) \end{array}$$

We obtain that $L_3 = \{3, 6\}$. Thus, $\phi_3 = x_3$ **xor** x_6.

We analyse the next d arrays:

$$\begin{array}{l} M^b_4 = (1 \quad 1 \quad 1) \\ M^b_5 = (1 \quad 1 \quad 1) \\ M^b_6 = (1 \quad 0 \quad 1) \\ M^b_7 = (1 \quad 1) \end{array}$$

We truncate the first 3 arrays and obtain that $L_4 = \{6\}$; since none of the truncated symbols is 0 we get $\phi_4 = x_6$.

When we analyse the next d arrays, we truncate the first 2 arrays and obtain that $L_5 = \{6\}$; since none of the truncated symbols is 0 we get $\phi_5 = x_6$.

Finally, for the last d arrays we only truncate the first array and obtain that $L_6 = \{6, 9\}$; since none of the truncated symbols is 0 we get $\phi_6 = x_6$ **xor** x_9.

Running the algorithm *Simplify* we obtain $F_1 = \{x_1\}$, $F_2 = F_4 = F_5 = F_7 = F_8 = F_9 = \emptyset$, $F_3 = \{x_3$ **xor** $x_6\}$ and $F_6 = \{x_6$ **xor** $x_9\}$.

Now running the algorithm *Filter* we obtain the set of equalities $U = \{x_1 = x_1^2, x_6 = x_6^2, x_3 = 0, x_9 = 0\}$ and we update the sets described above as follows $F_1 = \{x_1\}$, $F_2 = F_3 = F_4 = F_5 = F_7 = F_8 = F_9 = \emptyset$, $F_6 = \{x_6\}$.

The algorithm *Formula* produces $\phi_M = x_1 \wedge x_6$. A solution of the problem is having $x_1 = x_6 = 1$ and all the other variables equal to 0, and the replacement:

$$\begin{array}{ccccccccc} \diamond & b & a & d & c & \diamond & a & b & b \\ a & b & \diamond & d & c & e & a & b & \diamond \\ a & b & a & d & c & \diamond & & & \end{array}$$

Above, we wrote the arrays M_i as columns (with i increasing from left to right), the first element of the array being on the first line, the second element on the second line, and the third, when existing, on the third line.

Now, we move to an example for Problem 2. Consider $p = 6$, $d = 4$ and

$$w = bbadccabbdceabadcc.$$

From this word, we obtain exactly the sequence of arrays $M_1, \ldots, M_9$ depicted above. Thus, we will have $\phi_M = x_1 \wedge x_6$, and we have to set x_3 and x_9 to 0. However, we obtain the new set of equalities U' that contains the subset $\{\overline{x}_3 = x_3^1, \overline{x}_9 = x_9^1, \overline{x}_3^1 = x_9^1\}$. It is easy to see that, in this case, the problem has no solution.

For $d = 3$ one can easily obtain the solution $w = \diamond badc\diamond ab\diamond dceabadc\diamond$. After the replacements are applied, the arrays become:

$$\begin{array}{ccccccccc} \diamond & b & a & d & c & \diamond & a & b & \diamond \\ a & b & \diamond & d & c & e & a & b & a \\ a & b & a & d & c & \diamond & & & \end{array}$$

5. Conclusions

In this paper we solved efficiently a series of basic problems related to the concept of periodicity in the setting of partial words.

First, we presented a solution for the problem of identifying the periods of a partial word. Similar to the case of full words, our solution is based on preliminary string-matching step: we needed to find all the factors of a word compatible with the input word of our problem. Then, the information gathered in this step is used to actually find the periods. It is worth noting that the initial matching step is the most time consuming part of our algorithm; improving its running time would lead to an improvement of the running time of the whole algorithm. Thus, there are some possible ways to improve the performance of the algorithm we presented. The simplest one might be finding a direct algorithm that avoids this initial step and runs faster than the one we presented. Also, in the case when the partial words matching problem can be solved faster than $\mathcal{O}(n(\log\log n))$ we would canonically get an algorithm that runs in $\mathcal{O}(n(\log\log n))$ time. Note also that the matching problem we need to solve is not really general: the text has a quite restricted form, as it equals the pattern followed by a lengthy suffix that consists only of $\diamond$-symbols; it may be that such a problem can be solved faster than the general one, so this would improve the running time of our algorithm.

However, even in the case when the matching step is improved, we do not claim that one would reach an optimal algorithm. Indeed, we were not able to show any lower bounds on the time complexity of the problem of finding the periods of a partial word, except for the trivial linear time bound. It would be interesting to see such lower bounds for our problem, or, even more interesting, for the pattern matching problem for partial words.

Second, we discussed a problem where we produce in an efficient manner data structures that allow us to answer fast queries on the periodicity of the factors of a given partial word. The immediate question is whether we can construct faster (other) such data structures, from which we can obtain the answers to queries as fast. It may be worth noting that the data structures we construct are pretty close to oracles (that is, structures that already contain all the answers to possible queries). Maybe we can imagine structures that contain less information, can be built faster, but still preserve the query-answering time.

Nevertheless, the update operation defined in Problem 1 is an attempt to formulate and solve periodicity-related algorithmic problems in a dynamic (or, on-line) setting: the input is no longer completely given from the beginning, but is rather a stream of data. In this framework, it seems worth investigating the existence of efficient streaming algorithms solving similar problems: can we compute efficiently some succinct data-structures allowing us to retrieve facts regarding the periodicity of the factors of a partial word, provided that we do not have enough memory to store the entire word (and, in an even more restricted scenario, we can only read the word once)?

We also considered the problem of finding fast ways to substitute the letters of a full word by holes such that the word becomes periodic. In this case we propose an optimal solution, even for the case when we are interested in the minimal number substitution that lead to a periodic partial word. It seems interesting to us to see how efficiently we can solve problems of a similar fashion for a word's factors. More precisely, can we construct data structures that allow us answer fast to queries of the type: is there a d-valid p-periodic partial word compatible with a given factor of the input word? Or: what is the minimum number of holes that a d-valid p-periodic partial word compatible with a given factor of the input word has?

It is worth noting that our solution to Problem 2 relies heavily on the assumption that the whole input is known from the beginning. Related to the discussions already made for Problem 1, we might consider this problem, as well, in a dynamic framework. Can we still solve it when the word is given as a stream of data (and, basically, we have to compute the answer for any prefix of the word, and then update it when a new letter is given)? If yes, how efficiently can we do it? Is the problem still (efficiently) solvable when we lack enough memory to store the input word completely?

Finally, as a conclusion of this paper, we emphasise our view on the results we presented as steps towards developing a (yet, very incipient) tool-box of efficient algorithms solving problems on combinatoric properties of partial words. We contributed to this tool-box with a series of tools for testing properties related to the periodicity of partial words, but the problem whether these are the most efficient tools, as well as the problem of building new tools for other related questions remain open.

Acknowledgements

The work of Florin Manea is supported by the *DFG* grant 596676. The work of Robert Mercaş is supported by the *DFG* grant 582014.

References

[1] F. Manea, R. Mercaş, C. Tiseanu, Periodicity algorithms for partial words, in: F. Murlak, P. Sankowski (Eds.), Mathematical Foundations

of Computer Science 2011, Vol. 6907 of Lecture Notes in Computer Science, 2011, pp. 472–484.

[2] F. Blanchet-Sadri, Algorithmic Combinatorics on Partial Words, Chapman & Hall/CRC Press, 2008.

[3] C. Choffrut, J. Karhumäki, Combinatorics of words, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 1, Springer-Verlag, 1997, pp. 329–438.

[4] M. Lothaire, Combinatorics on Words, Cambridge University Press, 1997.

[5] Z. Galil, J. I. Seiferas, Time-space optimal string matching, Journal of Computer and System Sciences 26 (1983) 280–294.

[6] M. Crochemore, String-matching on ordered alphabets, Theoretical Computer Science 92 (1) (1992) 33–47.

[7] M. Crochemore, D. Perrin, Two-way string matching, Journal of the ACM 38 (3) (1991) 651–675.

[8] D. E. Knuth, J. H. Morris Jr., V. R. Pratt, Fast pattern matching in strings, SIAM Journal of Computing 6 (2) (1977) 323–350.

[9] F. Blanchet-Sadri, A. R. Anavekar, Testing primitivity on partial words, Discrete Applied Mathematics 155 (3) (2007) 279–287.

[10] F. Blanchet-Sadri, A. Chriscoe, Local periods and binary partial words: an algorithm, Theoretical Computer Science 314 (1-2) (2004) 189–216.

[11] R. Cole, R. Hariharan, Verifying candidate matches in sparse and wildcard matching, in: J. H. Reif (Ed.), 34th Annual ACM Symposium on Theory of Computing 2002, 2002, pp. 592–601.

[12] M. J. Fischer, M. S. Paterson, String matching and other products, in: R. Karp (Ed.), Complexity of Computation, Vol. 7 of SIAM-AMS Proceedings, 1974, pp. 113–125.

[13] P. Clifford, R. Clifford, Simple deterministic wildcard matching, Information Processing Letters 101 (2) (2007) 53–54.

[14] F. Manea, R. Mercaş, Freeness of partial words, Theoretical Computer Science 389 (1-2) (2007) 265–277.

[15] F. Blanchet-Sadri, R. Mercaş, A. Rashin, E. Willett, Periodicity algorithms and a conjecture on overlaps in partial words, Theoretical Computer Science 443 (2012) 35 – 45.

[16] A. Diaconu, F. Manea, C. Tiseanu, Combinatorial queries and updates on partial words, in: M. Kutylowski, W. Charatonik, M. Gebala (Eds.), 17th International Conference on Fundamentals of Computation Theory 2009, Vol. 5699 of Lecture Notes in Computer Science, 2009, pp. 96–108.

[17] P. Leupold, Languages of partial words - how to obtain them and what properties they have, Grammars 7 (2004) 179–192.

[18] P. Leupold, Partial words for DNA coding, in: G. Rozenberg, P. Yin, E. Winfree, J. H. Reif, B.-T. Zhang, M. H. Garzon, M. Cavaliere, M. J. Prez-Jimnez, L. Kari, S. Sahu (Eds.), 10th International Workshop on DNA Computing, Vol. 3384 of Lecture Notes in Computer Science, 2005, pp. 224–234.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press and McGraw-Hill Book Company, 2001.

[20] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, Journal of the ACM 53 (2006) 918–936.

[21] F. Blanchet-Sadri, C. D. Davis, J. Dodge, R. Mercaş, M. Moorefield, Unbordered partial words, Discrete Applied Mathematics 157 (5-6) (2009) 890 – 900.

[22] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1) (1981) 195–197.

[23] B. Aspvall, M. F. Plass, R. E. Tarjan, A linear-time algorithm for testing the truth of certain quantified boolean formulas, Information Processing Letters 8 (3) (1979) 121–123.