ELSEVIER

Theoretical Computer Science 262 (2001) 557–568

Theoretical Computer Science

www.elsevier.com/locate/tcs

Approximate periods of strings

Jeong Seop Sim[a,1], Costas S. Iliopoulos[b,c,2], Kunsoo Park[a,1,*], W.F. Smyth[c,d,3]

[a] *School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea*
[b] *Department of Computer Science, King's College London, London WC2R 2LS, UK*
[c] *School of Computing, Curtin University, Perth WA 6825, Australia*
[d] *Department of Computing & Software, McMaster University, Hamilton, Ontario, Canada L8S 4KI*

Received 27 December 1999; revised 27 June 2000; accepted 7 August 2000
Communicated by M. Crochemore

Abstract

The study of approximately periodic strings is relevant to diverse applications such as molecular biology, data compression, and computer-assisted music analysis. Here we study different forms of approximate periodicity under a variety of distance functions. We consider three related problems, for two of which we derive polynomial-time algorithms; we then show that the third problem is NP-complete.

Keywords: Periodicity; Approximate periods; Repetitions; Distance function

1. Introduction

Repetitive or periodic strings have been studied in such diverse fields as molecular biology, data compression, and computer-assisted music analysis. In response to requirements arising out of a variety of applications, interest has arisen in algorithms for finding *regularities* in strings; that is, periodicities of an approximate nature. Some important regularities that have been studied in the literature are the following:

- *Periods*: A string p is called a *period* of a string x if x can be written as $x = p^k p'$, where $k \geqslant 1$ and p' is a prefix of p. The shortest period of x is called *the period* of x. For example, if $x = abcabcab$, then abc, $abcabc$, and x itself are periods of x,

* Corresponding author. Fax: +82-2-886-7589.
E-mail addresses: jssim@theory.snu.ac.kr (J.S. Sim), csi@dcs.kcl.ac.uk (C.S. Iliopoulos), kpark@theory.snu.ac.kr (K. Park), smyth@mcmaster.ca (W.F. Smyth).

[1] Supported by KOSEF Grant 981-0925-128-2 and the Brain Korea 21 Project.
[2] Supported in part by the CCSLAAR Royal Society Research Grant.
[3] Supported by NSERC Grant No. A8180.

PII: S0304-3975(00)00365-0

while *abc* is *the* period of x. If x has a period p such that $|p| \leqslant |x|/2$, then x is said to be *periodic*. Further, if setting $x = p^k$ implies $k = 1$, x is said to be *primitive*; if $k \geqslant 2$, p^k is called a *repetition*.

- *Covers*: A string w is called a *cover* of x if x can be constructed by concatenations and superpositions of w. For example, if $x = ababaaba$, then *aba* and x are the covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*.
- *Seeds*: A substring w of x is called a *seed* of x if it is a cover of any superstring of x. For example, *aba* and *ababa* are some seeds of $x = ababaab$.
- *Repetitions*: A *repetition* is an immediately repeated nonempty string. For example, if $x = aababab$, then *aa* and *ababab* are repetitions in x; in particular, $a^2 = aa$ is called a *square* or *tandem repeat* and $(ab)^3 = ababab$ is called a *cube*.

The notions *cover* and *seed* are generalizations of periods in the sense that superpositions as well as concatenations are used to define them. A significant amount of research has been done on each of these four notions:

- *Periods*: The preprocessing of the Knuth–Morris–Pratt algorithm [21] finds all periods of x in linear time – in fact, all periods of every prefix of x. In parallel computation, Apostolico et al. [2] gave an optimal O(log log n) time algorithm for finding all periods, where n is the length of x.
- *Covers*: Apostolico et al. [4] introduced the notion of covers and described a linear-time algorithm to test whether x is superprimitive or not (see also [7, 8, 17]). Moore and Smyth [28] and recently Li and Smyth [24] gave linear-time algorithms for finding all covers of x. In parallel computation, Iliopoulos and Park [18] obtained an optimal O(log log n) time algorithm for finding all covers of x. Apostolico and Ehrenfeucht [3] and Iliopoulos and Mouchard [16] considered the problem of finding maximal quasiperiodic substrings of x. A two-dimensional variant of the covering problem was studied in [11, 14], and minimum covering by substrings of a given length in [19].
- *Seeds*: Iliopoulos et al. [15] introduced the notion of seeds and gave an O(n log n) time algorithm for computing all seeds of x. For the same problem, Berkman et al. [6] presented a parallel algorithm that requires O(log n) time and O(n log n) work.
- *Repetitions*: There are several O(n log n) time algorithms for finding all the repetitions in a string [10, 5, 26]. In parallel computation, Apostolico and Breslauer [1] gave an optimal O(log log n) time algorithm (i.e., total work is O(n log n)) for finding all the repetitions.

A natural extension of the repetition problems is to allow errors. Approximate repetitions are common in applications such as molecular biology and computer-assisted music analysis [9, 12]. Among the four notions above, only approximate repetitions have been studied. If $x = uww'v$ where w and w' are similar, ww' is called an *approximate square* or *approximate tandem repeat*. When there is a nonempty string y between w and w', we say that w and w' are an *approximate nontandem repeat*. In [23], Landau and Schmidt gave an O(kn log k log n) time algorithm for finding approximate squares whose edit distance is at most k in a text of length n. Schmidt also

gave an $O(n^2 \log n)$ algorithm for finding approximate tandem or nontandem repeats in [30] which uses an arbitrary score for similarity of repeated strings.

In this paper, we introduce the notion of *approximate periods* which is an approximate version of *periods*. Here we study different forms of approximate periodicity under a variety of distance functions. We consider three related problems, for two of which we derive polynomial-time algorithms; we then show that the third problem is NP-complete.

This paper is organized as follows. In Section 2, we describe some notations and definitions used in this paper. In Section 3, we define approximate periods and three related problems studied in this paper. Then we present polynomial-time algorithms for two problems and the proof of NP-completeness of the third problem in Section 4. We conclude in Section 5.

2. Preliminaries

A *string* is a sequence of zero or more characters from an alphabet Σ. The set of all strings over the alphabet Σ is denoted by Σ^* and all strings of length m over Σ is denoted by Σ^m. The empty string is denoted by ε. The length of a string x is denoted by $|x|$ and the ith character of x by $x[i]$. When a string w is $x[i]x[i+1]\cdots x[j]$, we denote w by $x[i..j]$ and w is called a *substring* (or a *factor* [13]) of x. Conversely, x is called a *superstring* of w. For example, *bcd* is a substring of *abcde*, and *abcde* is a superstring of *bcd*. A blank space is denoted by $\Delta \notin \Sigma$, and we regard it as a character for convenience.

A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$. A string w is a *subsequence* (also called a *subword* [13]) of x (or x is a *supersequence* of w) if w is obtained by deleting zero or more characters (at any positions) from x. For example, *ace* is a subsequence of *aabcdef*. For a given set S of strings, a string s is called a *common supersequence* of S if s is a supersequence of every string in S.

The *distance* $\delta(x, y)$ between two strings x and y is the minimum cost to convert one string x to the other string y. There are several well-known distance functions. The *edit distance* between two strings x and y is the minimum number of edit operations to convert x to y. The edit operations are the *insertion* of a character into x, the *deletion* of a character from x, and the *change* (or *substitution*) of a character in x with a character in y. Note that since we define the edit distance by the *number* of edit operations, the cost of each edit operation is 1. The *Hamming distance* between x and y is the smallest number of change operations to convert x to y. Note that the Hamming distance can be defined only when $|x| = |y|$ because it does not allow insertions and deletions. The edit distance can be generalized by using a penalty matrix. A *penalty matrix* specifies the substitution cost for each pair of characters and the insertion/deletion cost for each character. The *weighted edit distance* between x and y is the minimum cost to convert x to y using a penalty matrix. A penalty matrix is a metric when it satisfies

a	b	c	a	e
Δ	b	c	d	Δ
a	b	Δ	d	e

Fig. 1. An alignment.

the following conditions for all $a, b, c \in \Sigma \cup \{\Delta\}$:

- $\delta(a, b) \geqslant 0$,
- $\delta(a, b) = \delta(b, a)$,
- $\delta(a, a) = 0$, and
- $\delta(a, c) \leqslant \delta(a, b) + \delta(b, c)$ (triangle inequality).

An *alignment* of a set S of strings can be represented by a two-dimensional matrix, where each row is a string in S and all rows have the same length. The equality of lengths of all the rows can be obtained by inserting zero or more spaces to each of the strings in S. For example, when $S = \{abcae, bcd, abde\}$, an alignment of S is shown in Fig. 1.

We say that a distance function $\delta(x, y)$ is a *relative distance function* if the lengths of strings x and y are considered in the value of $\delta(x, y)$; otherwise, it is an *absolute distance function*. The Hamming distance and the edit distance are examples of absolute distance functions. There are two ways to define a relative distance between x and y:

- First, we can fix one of the two strings and define a relative distance function with respect to the fixed string. The *error ratio with respect to* x is defined to be $d/|x|$, where d is an absolute distance between x and y.
- Second, we can define a relative distance function symmetrically. The *symmetric error ratio* is defined to be d/l, where d is an absolute distance between x and y, and $l = (|x| + |y|)/2$ [31]. Note that we may take $l = |x| + |y|$ (then everything is the same except that the ratio is multiplied by 2).

If d is the edit distance between x and y, the error ratio with respect to x or the symmetric error ratio is called a *relative edit distance*. The weighted edit distance can also be used as a relative distance function because the penalty matrix can contain arbitrary costs.

3. Problem definitions

Given two strings x, p and a distance function δ, we define approximate periods as follows. If there exists a partition of x into disjoint blocks of substrings, i.e., $x = p_1 p_2 \cdots p_r$ ($p_i \neq \varepsilon$) such that $\delta(p, p_i) \leqslant t$ for $1 \leqslant i < r$, and $\delta(p', p_r) \leqslant t$ where p' is some prefix of p, we say that p is a *t-approximate period of* x (or p is an approximate period of x with distance t). Each p_i, $1 \leqslant i \leqslant r$, will be called a partition block of x. The last partition block p_r can be matched to any prefix p' of p as in the definition of exact periods. If p' is shorter than p, the last partition block will be

called *short*. Note that there can be several versions of approximate periods according to the definition of distance function δ.

We consider the following problems related to approximate periods.

Problem 1. *Given x, p, and any distance function δ, find the minimum t such that p is a t-approximate period of x.*

Since p is fixed in this case, it makes no difference whether δ is an absolute distance function or an error ratio with respect to p. If a threshold $k \leqslant |p|$ on the edit distance is given as input in Problem 1, the problem asks whether p is a k-approximate period of x or not. Note that if the edit distance is used for δ, it is trivially true that p is a $|p|$-approximate period of x.

Problem 2. *Given a string x and a relative distance function δ, find a substring p of x that is an approximate period of x with the minimum distance.*

Since the length of p is not (a priori) fixed in this problem, we need to use a relative distance function for δ (i.e., an error ratio or a weighted edit distance) rather than an absolute distance function. For example, if the absolute edit distance is used, every substring of x of length 1 is a 1-approximate period of x. Moreover, we assume that there are at least two partition blocks of x except possibly the short partition block, because otherwise the longest proper prefix of x (or any long prefix of x) can easily become an approximate period of x with a small distance. This assumption will be applied to Problem 3, too.

Problem 3. *Given a string x and a relative distance function δ, find a string p that is an approximate period of x with the minimum distance.*

This problem is harder than Problem 2 because p can be any string, not necessarily a substring of x.

4. Algorithms and NP-completeness

Basically, we will use weighted edit distances with metric penalty matrices for the distance function δ in each problem. Recall that a penalty matrix defines the substitution cost for each pair of characters and the insertion or deletion cost for each character.

4.1. Problem 1

Our algorithm for Problem 1 consists of two steps. Let $n = |x|$ and $m = |p|$.

(1) Let w_{ij} be the distance between p and $x[i..j]$, $1 \leqslant i \leqslant j < n$ and w_{in} be the minimum value of the distances between all prefixes of p and $x[i..n]$. Note that $x[i..n]$ can be matched to any prefix of p by the definition of approximate periods. Compute w_{ij} for all $1 \leqslant i \leqslant j \leqslant n$.

(2) Compute the minimum t such that p is a t-approximate period of x. We use dynamic programming to compute t as follows. Let t_i be the minimum value such that p is a t_i-approximate period of $x[1..i]$. Let $t_0 = 0$. For $i = 1$ to n, we compute t_i by the following formula:

$$t_i = \min_{0 \leqslant h < i} (\max(t_h, w_{h+1,i})).$$

The value t_n is the minimum t such that p is a t-approximate period of x.

To compute the distances in step 1, we use a dynamic programming table called the *D table*. To compute the distance between two strings x and y, a D table of size $(|x| + 1) \times (|y| + 1)$ is used. Each entry $D[i, j]$, $0 \leqslant i \leqslant |x|$ and $0 \leqslant j \leqslant |y|$, stores the minimum cost of transforming $x[1..i]$ to $y[1..j]$. Initially, $D[0,0] = 0$, $D[i,0] = D[i-1,0] + \delta(x[i], \Delta)$, and $D[0, j] = D[0, j-1] + \delta(\Delta, y[j])$. Then we can compute all the entries of the D table in $O(|x||y|)$ time by the following recurrence:

$$D[i, j] = \min \begin{cases} D[i-1, j] + \delta(x[i], \Delta) \\ D[i, j-1] + \delta(\Delta, y[j]) \\ D[i-1, j-1] + \delta(x[i], y[j]). \end{cases}$$

Note that Δ is a blank space, and thus $\delta(a, \Delta)$ means the deletion cost of a and $\delta(\Delta, a)$ means the insertion cost of a.

We now consider the time complexity of the above algorithm for Problem 1 under various distances functions. For a weighted edit distance, we make a D table of size $(m+1) \times (n-i+2)$ for each position i of x. Since the distances between all prefixes of p and $x[i..n]$ appear in the last column of the D table, we can get the minimum value of the last column as w_{in} without increasing the time complexity. Hence, step 1 takes $O(mn^2)$ time. In step 2, we can compute the minimum t in $O(n^2)$ time since we compare $O(n)$ values for each t_i. Therefore, the total time complexity is $O(mn^2)$.

When the edit distance is used for δ, this algorithm for Problem 1 can be improved. In this case, $\delta(a, b)$ is always 1 if $a \neq b$; $\delta(a, b) = 0$, otherwise. Now it is not necessary to compute the edit distances between p and the substrings of x whose lengths are larger than $2m$ because their edit distances with p will exceed m. Step 1 now takes $O(m^2 n)$ time since we make a D table of size $(m+1) \times (2m+1)$ for each position of x. Also, step 2 can be done in $O(mn)$ time since we compare $O(m)$ values at each position of x. Thus the time complexity is reduced to $O(m^2 n)$.

However, we can do better. Step 1 can be solved in $O(mn)$ time by the algorithm due to Landau et al. [22]. Given two strings x and y and a forward (resp. backward) solution for the comparison between x and y, the algorithm in [22] incrementally computes a solution for x and by (resp. yb) in $O(k)$ time, where b is an additional character and k is a threshold on the edit distance. This can be done due to the relationship between the solution for x and y and the solution for x and by. When $k = m$ (i.e., the threshold is not given), we can compute all the edit distances between p and every substring of x whose length is at most $2m$ in $O(mn)$ time using this algorithm. Recently, Kim and Park [20] gave a simpler $O(mn)$-time algorithm for the same problem. Therefore,

we can solve Problem 1 in $O(mn)$ time if the edit distance is used for δ. When the threshold k on the edit distance is given as input for Problem 1, it can be solved in $O(kn)$ time because each step of the above algorithm takes $O(kn)$ time.

If we use the Hamming distance for δ, the algorithm takes trivially $O(n)$ time since x must be partitioned into blocks of size m. (The last partition block can be shorter than m.) Therefore, we have the following theorem.

Theorem 1. *Problem* 1 *can be solved in* $O(mn^2)$ *time when a weighted edit distance is used for* δ. *If the edit distance* (*resp. the Hamming distance*) *is used for* δ, *it can be solved in* $O(mn)$ *time* (*resp. in* $O(n)$ *time*).

4.2. Problem 2

When a relative edit distance is used for δ, Problem 2 can be solved in $O(n^4)$ time by our algorithm for Problem 1. Let p be a candidate string for the approximate period of x. If we take each substring of x as p and apply the $O(mn)$ algorithm for Problem 1 (that uses the algorithm in [22]), it takes $O(|p|n)$ time for each p. Since there are $O(n^2)$ substrings of x, the overall time is $O(n^4)$.

Without using the somewhat complicated algorithm in [22], however, we can solve Problem 2 in $O(n^4)$ time by the following simple algorithm for weighted edit distances as well as relative edit distances. We take every substring of x as a candidate string p, and compute the minimum distance t such that p is a t-approximate period of x. But, we do this for all substrings of x that start at a position i at the same time, as follows.

Let T be the minimum distance so far. Initially, $T=\infty$. For $i=1$ to n, we do the following. For each i, we process the $n-i+1$ substrings that start at position i as candidate strings. Let m be the length of a chosen substring of x as p. Initially, $m=1$.

(1) Take $x[i..i+m-1]$ as p and compute w_{hj} for all $1 \leqslant h \leqslant j \leqslant n$. The definition of w_{hj} is the same as in Problem 1. This computation can be done by making n D tables with p and each of n suffixes of x. By adding just one row to each of previous D tables (i.e., n D tables when $p=x[i..i+m-2]$), we can compute these new D tables in $O(n^2)$ time. See Fig. 2. (Note that when $m=1$, we create new D tables.)

(2) Compute the minimum distance t such that p is a t-approximate period of x. This step is similar to the second step of the algorithm for Problem 1. Let t_j be the minimum value such that p is a t_j-approximate period of $x[1..j]$ and let $t_0=0$. For $j=1$ to n, we compute t_j by the following formula:

$$t_j = \min_{0 \leqslant h < j} (\max(t_h, w_{h+1,j})).$$

The value t_n is the minimum t such that p is a t-approximate period of x. If t_n is smaller than T, we update T with t_n. If $m < n-i+1$, increase m by 1 and go to step 1.

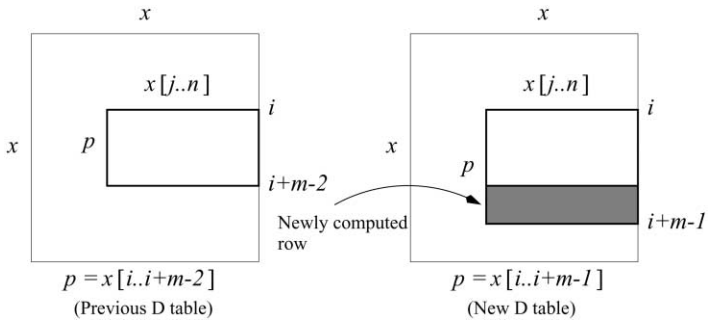

Fig. 2. Computing new D tables.

When all the steps are completed, the final value of T is the minimum distance and the substring p that is a T-approximate period of x is an answer to Problem 2.

Theorem 2. *Problem* 2 *can be solved in* $O(n^4)$ *time when a weighted edit distance or a relative edit distance is used for* δ. *When a relative Hamming distance is used for* δ, *Problem* 2 *can be solved in* $O(n^3)$ *time.*

Proof. For a weighted edit distance, we make n D tables in $O(n^2)$ time in step 1 and compute the minimum distance in $O(n^2)$ time in step 2. For $m=1$ to $n-i+1$, we repeat the two steps. Therefore, it takes $O(n^3)$ time for each i and the total time complexity of this algorithm is $O(n^4)$. If a relative edit distance is used, this algorithm can be slightly simplified as in Problem 1, but it still takes time $O(n^4)$.

For a relative Hamming distance, it takes $O(n)$ time for each candidate string and there are $O(n^2)$ candidate strings. Thus, the total time complexity is $O(n^3)$. □

4.3. Problem 3

Given a set of strings, the *shortest common supersequence* (SCS) problem is to find a shortest common supersequence of all strings in the set. The SCS problem is NP-complete [25, 29]. We will show that Problem 3 is NP-complete by a reduction from the SCS problem. In this section we will call Problem 3 *the AP problem* (abbreviation of the approximate period problem). The decision versions of the SCS and AP problems are as follows:

Definition 1. Given a positive integer m and a finite set S of strings from Σ^* where Σ is a finite alphabet, the SCS problem is to decide if there exists a common supersequence w of S such that $|w| \leqslant m$.

Definition 2. Given a number t, a string x from $(\Sigma')^*$ where Σ' is a finite alphabet, and a penalty matrix, the AP problem is to decide if there exists a string u such that u is a t-approximate period of x.

	0	*1*	*a*	*b*	$*_1$	$*_2$	#	$	Δ
0	0	2	1	2	2	2			1
1	2	0	2	1	2	2			1
a	1	2	0	2	1	1			1
b	2	1	2	0	1	1			1
$*_1$	2	2	1	1	0	2			2
$*_2$	2	2	1	1	2	0			2
#							0		
$								0	
Δ	1	1	1	1	2	2			0

Fig. 3. The penalty matrix M.

Now we transform an instance of the SCS problem to an instance of the AP problem. We can assume that $\Sigma = \{0, 1\}$ since the SCS problem is NP-complete even if $\Sigma = \{0, 1\}$ [27, 29]. Assume that there are n strings $s_1, \ldots, s_n$ in S. First, we set $\Sigma' = \Sigma \cup \{a, b, \#, \$, *_1, *_2, \Delta\}$. Let $x = \#*_1{}^m\$\ \#*_2{}^m\$\ \#s_1\$\ \#s_2\$ \cdots \#s_n\$$. Then, set $t = m$ and define the penalty matrix as in Fig. 3, where a shaded entry can be any value greater than m which makes the penalty matrix a metric. For example, every shaded entry can be $m + 1$. It is easy to see that this transformation can be done in polynomial time.

Lemma 1. *Assume that x is constructed as above. If u is an m-approximate period of x, then u is of the form $\#\alpha\$$ where $\alpha \in \{a, b\}^m$.*

Proof. We first show that u must have one # and one \$.

(1) Suppose that u has no # (resp. \$). Clearly, there exists a partition block of x which has at least one # (resp. \$), and the distance between u and the partition block is greater than m. Therefore, u must have at least one # and at least one \$.

(2) Suppose that u has more than one # (or \$). Assume that u has two #'s. (The other cases are similar.) Then u must also have two \$'s, because unless the number of #'s equals that of \$'s in u, at least one partition block of x cannot have the same numbers of #'s and \$'s to those of u. If the numbers of #'s and \$'s in u differ from those of a partition block of x, the distance between u and the partition block exceed m due to the penalty matrix M. Consider the first partition block of x. The first partition block is $\#*_1{}^m\$\ \#*_2{}^m\$$ because it must have two #'s and two \$'s as u. For the distance between u and the first partition block of x to be at most m, u must have at least m characters from $\{*_1, *_2\}$. In such cases, however, the distance between u and at least one partition block of x will exceed m.

It remains to show that $u = \#\alpha\$$ such that $\alpha \in \{a, b\}^m$. Since u has one # and one \$, x must be partitioned just after every occurrence of \$. Let u be of the form $\beta\#\alpha\$\gamma$, where $\beta, \alpha, \gamma \in \{0, 1, a, b, *_1, *_2, \Delta\}^*$. Consider the first two partition blocks $\#*_1{}^m\$$

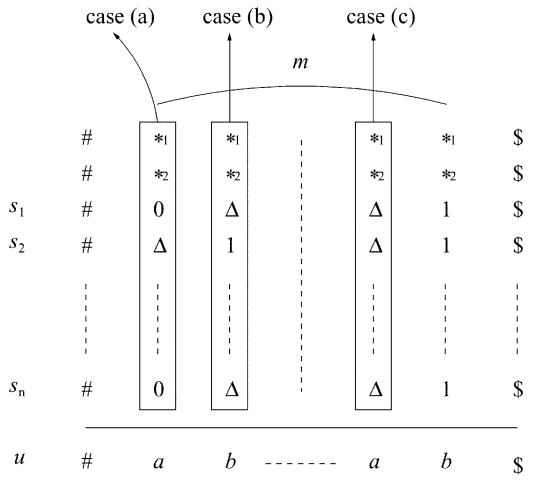

Fig. 4. An alignment of $S' \cup \{u\}$.

and $\#*_2{}^m\$$ of x. If α contains i $*_1$'s for $i \geqslant 1$, α must also have i $*_2$'s and the remaining $m-2i$ characters in α must be from $\{a, b\}$ so that the distances between u and the first two partition blocks of x do not exceed m. However, this makes the distance between u and any other partition block of x exceed m due to $*_1$'s and $*_2$'s in α. Hence α cannot have $*_1$ or $*_2$. Also, α cannot have any character from $\{0, 1, \Delta\}$ since $0, 1$ and Δ have cost 2 with $*_1$ and $*_2$ in the first two partition blocks of x. For the distances between u and the first two partition blocks of x to be at most m, β and γ must be empty and α must be of the form $\{a, b\}^m$. See Fig. 4. □

Theorem 3. *The AP problem is NP-complete.*

Proof. It is easy to see that the AP problem is in NP. To show that the AP problem is NP-complete, we need to show that S has a common supersequence w such that $|w| \leqslant m$ if and only if there exists a string u such that u is an m-approximate period of x.

(if) By Lemma 1, $u = \#\alpha\$$ where $\alpha \in \{a, b\}^m$. Since u is an m-approximate period of x, the distance between u and each partition block $\#s_i\$$ is at most m. (The distances between u and the first two partition blocks $\#*_1{}^m\$$ and $\#*_2{}^m\$$ are always m.) Consider an alignment of $S' \cup \{u\}$. Since $|\alpha| = m$ and the distance between α and s_i is at most m, each a (resp. b) in α must be aligned with 0 (resp. 1) or Δ in s_i. (See cases (a) and (b) in Fig. 4.) If we substitute 0 for a and 1 for b in α, we obtain a common supersequence w of $s_1, \ldots, s_n$ such that $|w| = m$. (Note that if a or b in α is aligned with Δ for all s_i, we can delete the character in α and we can obtain a common supersequence which is shorter than m. See case (c) in Fig. 4.) A similar alignment was used by Wang and Jiang [32].

(only if) Let w be a common supersequence of S such that $|w| \leqslant m$. Let α be the string constructed by substituting a for 0 and b for 1 in w. (When $|w| < m$, we append some characters from $\{a, b\}$ to α so that $|\alpha| = m$.) Partition x just after every occurrence of \$. The distance between each partition block of x and $\#\alpha\$$ is m since each a (resp. b) in α can be aligned with 0 (resp. 1), Δ, $*_1$, or $*_2$ in each partition block. Therefore, $\#\alpha\$$ is an m-approximate period of x. □

5. Conclusion

In this paper, we have introduced the notion of approximate periods and three related problems. We presented polynomial-time algorithms for two of the problems and an NP-completeness proof for the third problem.

There can be a variety of problems on approximate periods. For example, the following problem is an interesting one: Given a string x and a distance function, find a substring of x that has an approximate period. This is a problem of finding an interesting area (i.e., approximately periodic area) in a given string.

References

[1] A. Apostolico, D. Breslauer, An optimal $O(\log\log N)$-time parallel algorithm for detecting all squares in a string, SIAM J. Comput. 25 (6) (1996) 1318–1331.
[2] A. Apostolico, D. Breslauer, Z. Galil, Optimal parallel algorithms for periods, palindromes and squares, in: Proc. 19th Int. Colloq. Automata Languages and Programming, Lecture Notes in Computer Science, Vol. 623, Springer, Berlin, 1992, pp. 296–307.
[3] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theoret. Comput. Sci. 119 (2) (1993) 247–265.
[4] A. Apostolico, M. Farach, C.S. Iliopoulos, Optimal superprimitivity testing for strings, Inform. Process. Lett. 39 (1) (1991) 17–20.
[5] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, Theoret. Comput. Sci. 22 (1983) 297–315.
[6] O. Berkman, C.S. Iliopoulos, K. Park, The subtree max gap problem with application to parallel string covering, Inform. Comput. 123 (1) (1995) 127–137.
[7] D. Breslauer, An on-line string superprimitivity test, Inform. Process. Lett. 44 (1992) 345–347.
[8] D. Breslauer, Testing string superprimitivity in parallel, Inform. Process. Lett. 49 (5) (1994) 235–241.
[9] T. Crawford, C.S. Iliopoulos, R. Raman, String matching techniques for musical similarity and melodic recognition, Comput. Musicol. 11 (1998) 73–100.
[10] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Inform. Process. Lett. 12 (5) (1981) 244–250.
[11] M. Crochemore, C.S. Iliopoulos, M. Korda, Two-dimensional prefix string matching and covering on square matrices, Algorithmica 20 (1998) 353–373.
[12] M. Crochemore, C.S. Iliopoulos, H. Yu, Algorithms for computing evolutionary chains in molecular and musical sequences, Proc. 9th Austral. Workshop on Combinatorial Algorithms, 1998, pp. 172–185.
[13] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.
[14] C.S. Iliopoulos, M. Korda, Optimal parallel superprimitivity testing on square arrays, Parallel Process. Lett. 6 (3) (1996) 299–308.
[15] C.S. Iliopoulos, D.W.G. Moore, K. Park, Covering a string, Algorithmica 16 (1996) 288–297.
[16] C.S. Iliopoulos, L. Mouchard, An $O(n\log n)$ algorithm for computing all maximal quasiperiodicities in strings, in: Proc. Computing: Australasian Theory Symposium, Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 262–272.

[17] C.S. Iliopoulos, K. Park, An optimal $O(\log \log n)$-time algorithm for parallel superprimitivity testing, J. Korea Inform. Sci. Soc. 21 (1994) 1400–1404.
[18] C.S. Iliopoulos, K. Park, A work-time optimal algorithm for computing all string covers, Theoret. Comput. Sci. 164 (1996) 299–310.
[19] C.S. Iliopoulos, W.F. Smyth, On-line algorithms for k-covering, Proc. 9th Austral. Workshop on Combinatorial Algorithms, 1998, pp. 97–106.
[20] S. Kim, K. Park, A dynamic edit distance table, in: Proc. 11th Symp. Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1848, Springer, Berlin, 2000, pp. 60–68.
[21] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1) (1977) 323–350.
[22] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, SIAM J. Comput. 27 (2) (1998) 557–582.
[23] G.M. Landau, J.P. Schmidt, An algorithm for approximate tandem repeats, in: Proc. 4th Symp. Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 648, Springer, Berlin, 1993, pp. 120–133.
[24] Y. Li, W.F. Smyth, An optimal on-line algorithm to compute all the covers of a string, preprint.
[25] D. Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.
[26] M.G. Main R.J. Lorentz, An algorithm for finding all repetitions in a string, J. Algorithms 5 (1984) 422–432.
[27] M. Middendorf, More on the complexity of common superstring and supersequence problems, Theoret. Comput. Sci. 125 (2) (1994) 205–228.
[28] D. Moore, W.F. Smyth, A correction to An optimal algorithm to compute all the covers of a string, Inform. Process. Lett. 54 (2) (1995) 101–103.
[29] K.J. Räihä, E. Ukkonen, The shortest common supersequence problem over binary alphabet is NP-complete, Theoret. Comput. Sci. 16 (1981) 187–198.
[30] J.P. Schmidt, All highest scoring paths in weighted grid graphs and its application to finding all approximate repeats in strings, SIAM J. Comput. 27 (4) (1998) 972–992.
[31] P.H. Sellers, Pattern recognition genetic sequences by mismatch density, Bull. Math. Biol. 46 (4) (1984) 501–514.
[32] L. Wang, T. Jiang, On the complexity of multiple sequence alignment, J. Comput. Biol. 1 (4) (1994) 337–348.