Approximate Seeds of Strings

Manolis Christodoulakis[1] and Costas S. Iliopoulos[1] and
Kunsoo Park[2*] and Jeong Seop Sim[3]

[1] Department of Computer Science,
King's College London
e-mail: {manolis, csi}@dcs.kcl.ac.uk

[2] School of Computer Science and Engineering,
Seoul National University
e-mail: kpark@theory.snu.ac.kr

[3] Electronics and Telecommunications Research Institute
Daejeon 305-350, Korea
e-mail: simjs@etri.re.kr

Abstract. In this paper we study approximate seeds of strings, that is, substrings of a given string x that cover (by concatenations or overlaps) a superstring of x, under a variety of *distance* rules (the Hamming distance, the edit distance, and the weighted edit distance). We solve the *smallest distance approximate seed* problem and the *restricted smallest approximate seed* problem in polynomial time and we prove that the general *smallest approximate seed* problem is NP-complete.

Keywords: regularities, seeds, approximate seeds, Hamming distance, edit distance, weighted edit distance, penalty matrix.

1 Introduction

Finding *regularities* in strings is useful in a wide area of applications which involve string manipulations. Molecular biology, data compression and computer-assisted music analysis are classic examples. By regularities we mean repeated strings of an approximate nature. Examples of regularities include repetitions, periods, covers and seeds. Regularities in strings have been studied widely the last 20 years.

There are several $O(n \log n)$-time algorithms [11, 6, 27] for finding *repetitions*, that is, equal adjacent substrings, in a string x, where n is the length of x. Apostolico and Breslauer [2] gave an optimal $O(\log \log n)$-time parallel algorithm (i.e., total work is $O(n \log n)$) for finding all the repetitions.

The preprocessing of the Knuth-Morris-Pratt algorithm [22] finds all periods of x in linear time— in fact, all periods of every prefix of x. Apostolico, Breslauer and Galil [3] derived an optimal $O(\log \log n)$-time parallel algorithm for finding all periods.

*Work supported by IMT 2000 Project AB02, MOST grant M1-0309-06-0003, and Royal Society grant.

The fact that in practise it was often desirable to relax the meaning of "repetition", has led more recently to the study of a collection of related patterns—"covers" and "seeds". Covers are similar to periods, but now overlaps, as well as concatenations, are allowed. The notion of covers was introduced by Apostolico, Farach and Iliopoulos in [5], where a linear-time algorithm to test superprimitivity, was given (see also [8, 9, 18]). Moore and Smyth [29] and recently Li and Smyth [25] gave linear time-time algorithms for finding all covers of a string x. In parallel computation, Iliopoulos and Park [19] obtained an optimal $O(\log\log n)$ time algorithm for finding all covers of x. Apostolico and Ehrenfeucht [4] and Iliopoulos and Mouchard [17] considered the problem of finding maximal quasiperiodic substrings of x. A two-dimensional variant of the covering problem was studied in [12, 15], and a minimum covering by substrings of a given length in [20].

An extension of the notion of covers, is that of *seeds*; that is, covers of a superstring of x. The notion of seeds was introduced by Iliopoulos, Moore and Park [16] and an $O(n\log n)$-time algorithm was given for computing all seeds of x. A parallel algorithm for finding all seeds was presented by Berkman, Iliopoulos and Park [7], that requires $O(\log n)$ time and $O(n\log n)$ work.

In applications such as molecular biology and computer-assisted music analysis, finding exact repetitions is not always sufficient. A more appropriate notion is that of *approximate* repetitions ([10, 13]); that is, finding strings that are "similar" to a given pattern, by allowing errors. In this paper, we consider three different kinds of "similarity" (approximation): the *Hamming distance*, the *edit disctance* [1, 35] and a generalization of the edit distance, the *weighted edit distance*, where different costs are assigned to each substitution, insertion and deletion for each pair of symbols.

Approximate repetitions have been studied by Landau and Schmidt [24], who derived 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 [31] which uses an arbitrary score for similarity of repeated strings. More recently, Sim, Iliopoulos, Park and Smyth provided polynomial time algorithms for finding approximate periods [33] and, Sim, Park, Kim and Lee solved the approximate covers problem in [34].

In this paper, we introduce the notion of approximate seeds, an approximate version of seeds. We solve the *smallest distance approximate seed* problem and the *restricted smallest approximate seed* problem and we prove that the more general *smallest approximate seed* problem is NP-complete.

The paper is organized as follows. In section 2, we present some basic definitions. In section 3, we describe the notion of approximate seeds and we define the three problems studied in this paper. In section 4, we present the algorithms that solve the first two problems and the proof that the third problem is NP-complete. Section 5 contains our conclusion.

2 Preliminaries

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The length of a string x is denoted by $|x|$. The *empty string*, the string of length zero, is denoted by ε. The i-th symbol of a string x is denoted by $x[i]$.

A string w is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i..j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string w is a *prefix* of x if $x = wy$, for $y \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = yw$, for $w \in \Sigma^*$. We call a string w a *subsequence* (also called a subword [14]) of x (or x is a *supersequence* of w) if w is obtained by deleting zero or more symbols at any positions from x. For example, *ace* is a subsequence of *aabcde f*. For a given set S of strings, a string w is called a *common supersequence* of S if s is a supersequence of every string in S.

The string xy is a *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1..n]$ and $y = y[1..m]$ such that $x[n - i + 1..n] = y[1..i]$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x[1..n]y[i + 1..m]$ is said to be a *superposition* of x and y. Alternatively, we may say that x *overlaps* with y.

A substring y of x is called a *repetition* in x, if $x = uy^kv$, where u, y, v are substrings of x and $k \geq 2$, $|y| \neq 0$. For example, if $x = aababab$, then a (appearing in positions 1 and 2) and ab (appearing in positions 2, 4 and 6) are repetitions in x; in particular $a^2 = aa$ is called a *square* and $(ab)^3 = ababab$ is called a *cube*.

A substring w is called a *period* of a string x, if x can be written as $x = w^kw'$ where $k \geq 1$ and w' is a prefix of w. The shortest period of x is called *the period* of x. For example, if $x = abcabcab$, then abc, $abcabc$ and the string x itself are periods of x, while abc is *the* period of x.

A substring w of x is called a *cover* of x, if x can be constructed by concatenating or overlapping copies of w. We also say that w *covers* x. For example, if $x = ababaaba$, then aba and x are covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*.

A substring w of x is called a *seed* of x, if w covers one superstring of x (this can be any superstring of x, including x itself). For example, aba and $ababa$ are some seeds of $x = ababaab$.

We call the *distance* $\delta(x, y)$ between two strings x and y, the minimum cost to transform one string x to the other string y. There are several well known distance functions, described in the next paragraph. The special symbol Δ is used to represent the absence of a character.

2.1 Distance functions

The *edit distance* between two strings is the minimum number of *edit operations* that transform one string into another. The edit operations are the *insertion* of an extraneous symbol (e.g., $\Delta \to a$), the *deletion* of a symbol (e.g., $a \to \Delta$) and the *substitution* of a symbol by another symbol (e.g., $a \to b$). Note that in the edit distance model we only count the *number* of edit operations, considering the cost of each operation equal to 1.

The *Hamming distance* between two strings is the minimum number of *substitutions* (e.g., $a \to b$) that transform one string to the other. Note that the Hamming distance can be defined only when the two strings have the same length, because it does not allow insertions and deletions.

We also consider a generalized version of the edit distance model, the *weighted edit distance*, where the edit operations no longer have the same costs. It makes use

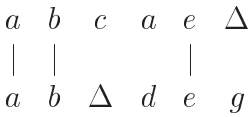

Figure 1: Alignment example

of a *penalty matrix*, a matrix that specifies the cost of each substitution for each pair of symbols, and the insertion and deletion cost for each character. A penalty matrix is a metric when it satisfies the following conditions for all $a, b, c \in \Sigma \cup \{\Delta\}$:

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

The similarity between two strings can be seen by using an *alignment*; that is, any pairing of symbols subject to the restriction that if lines were drawn between paired symbols, as in Figure 1, the lines would not cross. The equality of the lengths can be obtained by inserting or deleting zero or more symbols. In our example, the string "abcae" is transformed to "abdeg" by deleting, substituting and inserting a character at positions 3, 4 and 6, respectively. Note that this is not the only possible alignment between the two strings.

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 $l = (|x| + |y|)/2$ [32]. Note that we may take $l = |x| + |y|$, in which case 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

Definition 1 Let x and s be strings over Σ^*, δ be a distance function and t be a number. We call s a t-approximate seed of x if and only if there exist strings $s_1, s_2, \ldots, s_r$ $(s_i \neq \varepsilon)$ such that

(i) $\delta(s, s_i) \leq t$, for $1 \leq i \leq r$, and

(ii) there exists a superstring $y = uxv$, $|u| < |s|$ and $|v| < |s|$, of x that can be constructed by overlapping or concatenating copies of the strings $s_1, s_2, \ldots, s_r$.

Each s_i, $1 \leq i \leq r$, will be called a seed block of x.

Note that y can be *any* superstring of x, including x itself (in which case, s is an approximate cover). Note, also, that there can be several versions of approximate seeds according to the definition of distance function δ.

An example of an approximate seed is shown in Figure 2. For strings $x = BABACCB$ and $s = ABAB$, s is an approximate seed of x with error 1 (hamming distance), because there exist the strings $s_1 = ABAB, s_2 = ABAC, s_3 = CBAB$, such that the distance between s and each s_i is no more than 1, and by concatenating or overlapping the strings s_1, s_2, s_3 we construct a superstring of x, $y = ABABACCBAB$.

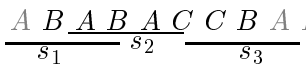

Figure 2: Approximate Seed example.

We consider the following three problems related to approximate seeds.

Problem 1 Smallest Distance Approximate Seed *Let x be a string of length n, s be a string of length m, and δ be a distance function. Find the minimum number t such that s is a t-approximate seed of x.*

In this problem, the string s is given a priori. Thus, it makes no difference whether δ is an absolute distance function or an error ratio with respect to s. If a threshold $k \leq |s|$ on the edit distance is given as input to Problem 1, the problem asks whether s is a k-approximate seed of x or not (the *k-approximate seed* problem). Note that if the edit distance is used for δ, it is trivially true that s is an $|s|$-approximate seed of x.

Problem 2 Restricted Smallest Approximate Seed *Given a string x of length n, find a substring s of x such that: s is a t-approximate seed of x and there is no substring of x that is a k-approximate seed of x for all $k < t$.*

Since any substring of x can be a candidate for s, the length of s is not (a priori) fixed in this problem. Therefore, we need to use a relative distance function (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 seed of x. Moreover, we assume that s is of length at most $|x|/2$, because, otherwise the longest proper prefix of x (or any long prefix of x) can easily become an approximate seed of x with a small distance. This assumption will be applied to Problem 3, too.

Problem 3 SMALLEST APPROXIMATE SEED *Given a string x of length n, find a string s such that: s is a t-approximate seed of x and there is no substring of x that is a k-approximate seed of x for all $k < t$.*

Problem 3 is a generalization of Problem 2; s can now be any string, not necessarily a substring of x. Obviously, this problem is harder than the previous one; we will prove that it is NP-complete.

4 Algorithms and NP-Completeness

4.1 Problem 1

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

1. *Compute the distance between s and every substring of x.*

   We denote by w_{ij} the distance between s and $x[i..j]$, for $1 < i \leq j < n$. Note that, by definition of approximate seeds, $x[i..n]$ can be matched to any prefix of s, and $x[1..j]$ can be matched to any suffix of s (because s has to cover *any* superstring of x). Thus, we denote w_{in} the minimum value of the distances between all prefixes of s and $x[i..n]$, and w_{1j} the minimum value of the distances between all suffixes of s and $x[1..j]$.

2. *Compute the minimum t such that s is a t-approximate seed of x.*

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

   $$t_i = \min_{0 \leq h < i} \{\max\{\min_{h \leq j < i} \{t_j\}, w_{h+1,i}\}\} \tag{1}$$

   The value t_n is the minimum t such that s is a t-approximate seed of x.

To compute the distance between two strings, x and y, in step 1, a dynamic programming table, called the D *table*, of size $(|x|+1) \times (|y|+1)$, is used. Each entry $D[i,j]$, $0 \leq i \leq |x|$ and $0 \leq j \leq |y|$, 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:

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

where $\delta(a,b)$ is the cost of substituting character a with character b, $\delta(a,\Delta)$ is the cost of deleting a and $\delta(\Delta,a)$ is the cost of inserting a.

The second step of the algorithm is computed as shown in Figure 3. For every h, we cover $x[h+1..i]$ with one copy of s, with error $w_{h+1,i}$. What is left to be covered is $x[1..h]$. We obtain this by covering either $x[1..h]$, with error $t[h]$, or $x[1..h+1]$, with error $t[h+1]$, ... or $x[1..i-1]$, with error $t[i-1]$, (in general $x[1..j]$, with error $t[j]$); we choose the $x[1..j]$ (the shaded box) that gives the smallest error. Note that, this box covers a superstring of $x[1..j]$.

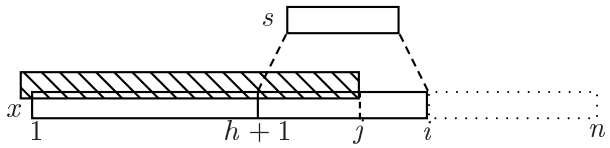

Figure 3: The second step of the algorithm.

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

PROOF. For an arbitrary penalty matrix, step 1 takes $O(mn^2)$ time, since we make a D table of size $(m+1) \times (n-i+2)$ for each position i of x. The fact that a *superstring* of x, rather than x itself, has to be "covered" does not increase the time complexity, if we use the following procedure: instead of computing a new D-table between each $s[1..k]$ (resp. $s[k..m]$) and $x[i..n]$ (resp. $x[1..j]$), we just make one D-table between s and $x[i..n]$ (resp. s^R $(x[1..j])^R$) and take the minimum value of the last column of this table.

In step 2, we can compute the minimum t in $O(n^2)$ time as follows. The inner *min* loop of formula (1) can be computed in constant time by reusing the *min* values computed in the previous round. The outer *min* loop is repeated i times, for $1 \leq i \leq n$, i.e., $O(n^2)$ repetitions.

Thus, the total time complexity is $O(mn^2)$.

When the edit distance is used for the measure of similarity, this algorithm for Problem 1 can be improved. In this case, $\delta(a, b)$ is always 1 if $a \neq b$ and $\delta(a, b) = 0$ otherwise. Now it is not necessary to compute the edit distances between s and the substrings of x whose lengths are larger than $2m$ because their edit distances with s will exceed m. (It is trivially true that s is an m-approximate seed of x.) Step 1 now takes $O(m^2n)$ 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 $O(m^2n)$.

However, we can do better. Step 1 can be solved in $O(mn)$ time by the algorithm due to Landau, Myers and Schmidt [23]. Given two strings x and y and a forward (resp. backward) solution for the comparison between x and y, the algorithm in [23] 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 s and every substring of x whose length is at most $2m$ in $O(mn)$ time using this algorithm. Recently, Kim and Park [21] 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 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, in step 1 we consider only the substrings of x of length m. (Recall that the Hamming distance is defined only between strings of equal length) Since there are $O(n)$ such substrings, and we need $O(m)$ time to compute the distance between each substring and s, step 1 takes $O(mn)$ time. Also, as in the case of the edit distance, step 2 can be done in $O(mn)$ time (we compare $O(m)$ values at each position of x). Thus, the overall time complexity is $O(mn)$. □

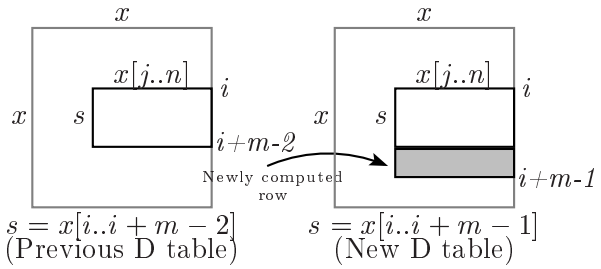

Figure 4: Computing new D tables

4.2 Problem 2

In this problem, we are not given a string s. Any substring of x is now a candidate for approximate seed. Let s be such a candidate string. Recall that, since the length of s is not fixed in this case, we need to use a relative distance function (rather than an absolute distance function); that is, an error ratio, in the case of the Hamming or edit distance, or a weighted edit distance.

When the relative edit distance is used for the measure of similarity, Problem 2 can be solved in $O(n^4)$ time by our algorithm for Problem 1. If we take each substring of x as s and apply the $O(mn)$ algorithm for Problem 1 (that uses the algorithm in [23]), it takes $O(|s|n)$ time for each s. Since there are $O(n^2)$ substrings of x, the overall time is $O(n^4)$.

For weighted edit distances (as well as for relative edit distances), we can solve Problem 2 in $O(n^4)$ time, without using the somewhat complicated algorithm in [23]. Like before, we consider every substring of x as candidate string s, and we solve Problem 1 for x and s. But, we do this, by processing all the substrings of x that start at position i, at the same time, as follows.

Let T be the minimum distance so far. Initially, $T = \infty$. For each i, $1 \leq i \leq n$, 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 s. Initially, $m = 1$.

1. Take $x[i..i + m - 1]$ as s and compute w_{hj}, for all $1 \leq h \leq j \leq n$. This computation can be done by making n D tables with s and each of the n suffixes of x. By adding just one row to each of previous D tables (i.e., n D tables when $s = x[i..i + m - 2]$), we can compute these new D tables in $O(n^2)$ time. See Figure 4. (Note that when $m = 1$, we create new D tables.)

2. Compute the minimum distance t such that s is a t-approximate seed of x. This step is similar to the second step of the algorithm for Problem 1. Let t_i be the minimum value such that s is a t_i-approximate seed of $x[1..i]$ and $t_0 = 0$. For $i = 1$ to n, we compute t_i by the following formula:

$$t_i = \min_{0 \leq h < i} \{\max\{\min_{h \leq j < i} \{t_j\}, w_{h+1,i}\}\}$$

   The value t_n is the minimum t such that s is a t-approximate seed 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.

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

(Note that there can be more than one substring s that are T-approximate seeds of x).

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, the algorithm can be slightly simplified, as in Problem 1, but it still takes $O(n^4)$ time.

For a relative Hamming distance, it takes $O(n)$ time for each candidate string and since there are $O(n^2)$ candidate strings, 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 [26, 30]. 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 *SAS problem* (abbreviation of the smallest approximate seed problem). The decision versions of the SCS and SAS problems are as follows:

Definition 2 (SCS) 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 $|w| \leq m$.

Definition 3 (SAS) Given a number t, a string x from $(\Sigma')^*$ where Σ' is a finite alphabet, and a penalty matrix, the SAS problem is to decide if there exists a string u such that u is a t-approximate seed of x.

Theorem 3 The SAS problem is NP-complete.

5 Conclusions

In this paper, we solved the *smallest distance approximate seed* problem, in $O(mn)$ time for the Hamming and edit distance and $O(mn^2)$ for the weighted edit distance, and the *restricted smallest approximate seed* problem, in $O(n^4)$ time for the edit and weighted edit distance and $O(n^3)$ for the Hamming distance. We also proved that the *smallest approximate seed* problem is NP-complete.

The significance of our work comes from the fact that we solved the first two problems for approximate seeds, with exactly the same time complexities as those for approximate periods [33] and approximate covers [34], despite the fact that seeds allow overlaps, as well as concatenations, and cover a *superstring* of a string x (rather than covering the string x itself).

References

[1] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[2] A. Apostolico and D. Breslauer. An optimal $O(\log \log N)$-time parallel algorithm for detecting all squares in a string. *SIAM Journal on Computing*, 25(6):1318–1331, 1996.

[3] A. Apostolico, D. Breslauer, and Z. Galil. Optimal parallel algorithms for periods, palindromes and squares. *Proc. 19th Int. Colloq. Automata Languages and Programming*, 623:296–307, 1992.

[4] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.

[5] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.

[6] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.

[7] O. Berkman, C. S. Iliopoulos, and K. Park. The subtree max gap problem with application to parallel string covering. *Information and Computation*, 123(1):127–137, 1995.

[8] D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.

[9] D. Breslauer. Testing string superprimitivity in parallel. *Information Processing Letters*, 49(5):235–241, 1994.

[10] T. Crawford, C. S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.

[11] M. Crochemore. An optimal algorithm for computing repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[12] M. Crochemore, C. S. Iliopoulos, and M. Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20:353–373, 1998.

[13] M. Crochemore, C. S. Iliopoulos, and H. Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In *Proc. 9th Australasian Workshop on Combinatorial Algorithms*, pages 172–185, 1998.

[14] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[15] C. S. Iliopoulos and M. Korda. Optimal parallel superprimitivity testing on square arrays. *Parallel Processing Letters*, 6(3):299–308, 1996.

[16] C. S. Iliopoulos, D. Moore, and K. Park. Covering a string. *Algorithmica*, 16:288–297, 1996.

[17] C. S. Iliopoulos and L. Mouchard. An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings. In *Proc. Computing: Australasian Theory Symposium*, pages 262–272. Lecture Notes in Computer Science, 1999.

[18] C. S. Iliopoulos and K. Park. An optimal $O(\log \log n)$-time algorithm for parallel superprimitivity testing. *J. Korea Inform. Sci. Soc.*, 21:1400–1404, 1994.

[19] C. S. Iliopoulos and K. Park. A work-time optimal algorithm for computing all string covers. *Theoretical Computer Science*, 164:299–310, 1996.

[20] C. S. Iliopoulos and W. F. Smyth. On-line algorithms for k-covering. In *Proceedings of the 9th Australasian Workshop On Combinatorial Algorithms*, pages 97–106, Perth, WA, Australia, 1998.

[21] S. Kim and K. Park. A dynamic edit distance table. In *Proc. 11th Symp. Combinatorial Pattern Matching*, volume 1848, pages 60–68. Springer, Berlin, 2000.

[22] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern mathcing in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[23] G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[24] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, number 684, pages 120–133, Padova, Italy, 1993. Springer-Verlag, Berlin.

[25] Y. Li and W. F. Smyth. An optimal on-line algorithm to compute all the covers of a string.

[26] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[27] M. G. Main and R. J. Lorentz. An algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–532, 1984.

[28] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125(2):205–228, 1994.

[29] D. Moore and W. F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995.

[30] K. J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981.

[31] J. P. Schmidt. All highest scoring paths in weighted grid graphs and its application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[32] P. H. Sellers. Pattern recognition genetic sequences by mismatch density. *Bulletin of Mathematical Biology*, 46(4):501–514, 1984.

[33] J. S. Sim, C. S. Iliopoulos, K. Park, and W. F. Smyth. Approximate periods of strings. *Theoretical Computer Science*, 262:557–568, 2001.

[34] J. S. Sim, K. Park, S. Kim, and J. Lee. Finding approximate covers of strings. *Journal of Korea Information Science Society*, 29(1):16–21, 2002.

[35] R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.