Approximate Periods of Strings

Jeong Seop Sim[1] ⋆ Costas S. Iliopoulos [2 4] †
Kunsoo Park [1] ⋆ W. F. Smyth [3 4] ‡

[1] Department of Computer Engineering, Seoul National University
[2] Department of Computer Science, King's College London
[3] Department of Computing & Software, McMaster University
[4] School of Computing, Curtin University

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 rules. We consider three related problems, for two of which we derive polynomial-time algorithms; we then show that the third problem is NP-complete.

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 $k \geq 1$ 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 are periods of x, while abc is *the* period of x. If x has a period p such that $|p| \leq |x|/2$, then x is said to be *periodic*. Further, if setting $x = p^k$ implies $k = 1$, x is said to be *primitive*; if $k \geq 2$, 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 substring w of x that is a repetition is called a *repetition* or *tandem repeat in* x. For example, if $x = aababab$, then aa and $ababab$ are

⋆ {jssim, kpark}@theory.snu.ac.kr. Supported by KOSEF Grant 981-0925-128-2.
† csi@dcs.kcl.ac.uk. Supported in part by the CCSLAAR Royal Society Researh Grant.
‡ smyth@mcmaster.ca. Supported by NSERC Grant No. A8180.

repetitions in x; in particular, $a^2 = aa$ is called a *square* 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 [19] finds all periods of x in linear time — in fact, all periods of every prefix of x. In parallel computation, Apostolico, Breslauer and Galil [2] gave an optimal $O(\log\log n)$ time algorithm for finding all periods, where n is the length of x.
- **Covers:** Apostolico, Farach and Iliopoulos [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 [26] and recently Li and Smyth [22] gave linear-time algorithms for finding all covers of x. In parallel computation, Iliopoulos and Park [16] obtained an optimal $O(\log\log n)$ time algorithm for finding all covers of x. Apostolico and Ehrenfeucht [3] and Iliopoulos and Mouchard [15] considered the problem of finding maximal quasiperiodic substrings of x. A two-dimensional variant of the covering problem was studied in [11, 13], and minimum covering by substrings of given length in [18].
- **Seeds:** Iliopoulos, Moore and Park [14] 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, Iliopoulos and Park [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, 24]. 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 [21], Landau and Schmidt gave an $O(kn \log k \log n)$ time algorithm for finding repeated patterns 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 [28] which uses an arbitrary score for similarity of repeated strings.

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

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^*. The empty string is denoted by ϵ. The ith character of a string x is denoted by $x[i]$. A substring of x that starts at position i and ends at position j is denoted by $x[i..j]$.

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* 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 $aabcdef$.

2.1 Measures

ABSOLUTE MEASURES. To measure the similarity (or distance) between two strings, the Hamming distance and the edit distance are widely used. The *Hamming distance* between two strings x and y is defined to be the smallest number of *change* operations to convert x to y. The *edit distance* is defined to be the smallest number of *change*, *insert*, and *delete* operations to convert x to y. In more general cases, especially in molecular biology, a penalty matrix is used. A penalty matrix specifies the substitution cost for each pair of characters and the insertion/deletion cost for each character. An arbitrary penalty matrix can also be used as a relative measure because it can contain both positive and negative costs [28]. It is common to assume that a penalty matrix satisfies the triangle inequality [30].

RELATIVE MEASURES. When we want to compare the similarity between x and y and the similarity between x' and y', we need relative measures (rather than absolute measures) because the lengths of the strings x, y, x', y' may be different. There are two ways to define relative measures between x and y:

- First, we can fix one of the two strings and define a relative measure with respect to the fixed string. The *error ratio with respect to* x is defined to be $t/|x|$, where t is an absolute measure between x and y.
- Second, we can define a relative measure symmetrically. The *symmetric error ratio* is defined to be t/l, where t is an absolute measure between x and y, and $l = (|x| + |y|)/2$ [29]. Note that we may take $l = |x| + |y|$ (then everything is the same except that the ratio is multiplied by 2).

3 Problem Definitions

Given two strings x and p, we define approximate periods as follows. If there exists a partition of x into disjoint blocks of substrings, i.e., $x = p_1 p_2 \cdots p_r$ ($p_i \neq \epsilon$) such that the distance between p and p_i for every $1 \leq i \leq r$ is less than or equal to t, 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, $1 \leq i \leq r$, will be called a partition block of x. Note that there can be several versions of approximate periods according to the definition of *distance*. This definition of approximate periods can be considered as an approximate version of the three notions *periods*, *covers*, and *seeds* discussed above, because

(i) superpositions in defining covers and seeds and
(ii) extra characters at the ends of a given string in defining periods and seeds

can be accounted for in some degree when we use edit distances for the measure. Of course, if we allow overlaps between p_i's, then we could extend the definition of an approximate period. But this will merely increase the complexity of problems of finding approximate periods.

We consider the following problems related to approximate periods.

Problem 1. Given x and p, 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 we use the absolute Hamming (or edit) distance or the error ratio with respect to p. We can also use a penalty matrix for the measure. If a threshold k 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.

Problem 2. Given a string x, 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 relative measures (i.e., error ratios or penalty matrices) rather than absolute measures.

Problem 3. Given a string x, 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 arbitrary penalty matrices for the measure of similarity 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. Compute the distance between p and every substring of x.

2. Compute the minimum t such that p is a t-approximate period of x. We use dynamic programming to compute t. Let w_{ij} be the distance between p and $x[i..j]$. These values of w_{ij} are obtained from the first step. 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:

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

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 the 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 $D[i,j]$ $(0 \le i \le |x|, 0 \le j \le |y|)$ 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 transforming the character a to b. (Δ is a space, so $\delta(a,\Delta)$ means the deletion cost of a and $\delta(\Delta, a)$ means the insertion cost of a.)

Theorem 1. *Problem 1 can be solved in $O(mn^2)$ time when an arbitrary penalty matrix is used for the measure of similarity. If the edit distance (resp. the Hamming distance) is used for the measure, it can be solved in $O(mn)$ time (resp. in $O(n)$ time).*

Proof. For an arbitrary penalty matrix, step 1 takes $O(mn^2)$ time since we make a D table of size $m \times (n-i+1)$ for each position i of x. In step 2, we can compute the minimum t in $O(n^2)$ time since we compare $O(n)$ values at each position of x. 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 $a \ne b$; $\delta(a,b) = 0$, 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. (It is trivially true that p is an m-approximate period of x.) Step 1 now takes $O(m^2n)$ time since we make a D table of size $m \times 2m$ 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 [20]. Given two strings x and y and a forward (resp. backward) solution for the comparison between x and y, the algorithm in [20] 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. Therefore, we can solve Problem 1 in $O(mn)$ time if the edit distance is used for the measure of similarity.

If we use the Hamming distance for the measure, it takes trivially $O(n)$ time since x must be partitioned into blocks of size m. □

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.

4.2 Problem 2

Let p be a candidate string for the approximate period of x. If the Hamming (or edit) distance is used for Problem 2, we need to use relative measures because the length of p varies. (If the absolute Hamming or edit distance is used, every substring of x of length 1 is a 1-approximate period of x.) We can use the *error ratio* t/l for the measure of similarity, where t is the Hamming (or edit) distance between the two strings and l is either the average length of the two strings (symmetric error ratio) or the length of p (error ratio with respect to p).

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 p and apply the $O(mn)$ algorithm for Problem 1 (that uses the algorithm in [20]), 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 [20], however, we can solve Problem 2 in $O(n^4)$ time by the following simple algorithm for arbitrary penalty matrices.

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

1. Take $x[i..i+m-1]$ as p and compute the distance between p and every substring of x. This 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. (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 w_{hj} be the distance between p and $x[h..j]$ which is obtained from step 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:

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

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

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

Theorem 2. *Problem 2 can be solved in $O(n^4)$ time when an arbitrary penalty matrix is used for the measure of similarity. If the Hamming distance is used for the measure, it can be solved in $O(n^3)$ time.*

Proof. For an arbitrary penalty matrix, 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 the relative edit distance is used, this algorithm can be slightly simplified as in Problem 1, but it still takes time $O(n^4)$.

If the relative Hamming distance is used for the measure, Problem 2 can be solved in $O(n^3)$ time because there are $O(n^2)$ candidates for p and $O(n)$ time is required for each candidate. □

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 [23, 27]. 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 string w with $|w| \leq m$ such that w is a supersequence of each string in S.*

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.*

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 $\Sigma = \{0, 1\}$ [25, 27]. First, we set $\Sigma' = \Sigma \cup \{a, b, \#, \$, *_1, *_2, \Delta\}$. Assume that there are n strings $s_1, \ldots, s_n$ in S. Let $x = \# s_1 \$ \# s_2 \$ \cdots \# s_n \$ \# {*_1}^m \$ \# {*_2}^m \$$. Then, set $t = m$ and define the penalty matrix as in Figure 1, where a shaded entry can be any value greater than m. It is easy to see that this transformation can be done in polynomial time. Note that the penalty matrix M is a metric.

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 $\alpha \in \{a, b\}^m$.*

	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. 1. The penalty matrix 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 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. Consider the last partition block of x. Since the last block must have two #'s and two \$'s as u, it contains $\#{*_1}^m\$\#{*_2}^m\$$. For the distance between u and the last 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 any other partition block of x will exceed m.

It remains to show that $u = \#\alpha\$$ where $\alpha \in \{a, b\}^m$. 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 last two blocks $\#{*_1}^m\$$ and $\#{*_2}^m\$$ of x. If α contains i $*_1$'s for $i \geq 1$, 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 last two 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 last two blocks of x. For the distances between u and the last two blocks of x to be at most m, β and γ must be empty and α must be of the form $\{a, b\}^m$. $\square$

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 $|w| \leq m$ 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 last two blocks $\#{*_1}^m\$$ and $\#{*_2}^m\$$ are always m.) Since $|\alpha| = m$ and the distance between α and s_i is at most m, each 0 (resp. 1) in s_i must be aligned with a (resp. b) in α. That is, each a (resp. b) in α must be aligned with 0 (resp. 1) or Δ in s_i. 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.) A similar alignment was used by Wang and Jiang [30].

(only if) Let s be a common supersequence of S such that $|s| \leq m$. Let α be the string constructed by substituting a for 0 and b for 1 in s. Partition x just after every occurrence of \$. The distance between each partition block of x and $\#\alpha\$$ is at most 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. □

References

1. 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 (1996), 1318-1331.
2. A. Apostolico, D. Breslauer and Z. Galil, Optimal parallel algorithms for periods, palindromes and squares, *Proc. 19th Int. Colloq. Automata Languages and Programming*, Lecture Notes in Computer Science 623 (1992), 296-307.
3. A. Apostolico and A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, *Theoretical Computer Science* 119, 2(1993), 247-265.
4. A. Apostolico, M. Farach and C.S. Iliopoulos, Optimal superprimitivity testing for strings, *Information Processing Letters* 39, 1 (1991), 17-20.
5. A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, *Theoretical Computer Science* 22, (1983), 297-315.
6. O. Berkman, C.S. Iliopoulos and K. Park, The subtree max gap problem with application to parallel string covering, *Information and Computation* 123, 1 (1995), 127-137.
7. D. Breslauer, An on-line string superprimitivity test, *Information Processing Letters* 44 (1992), 345-347.
8. D. Breslauer, Testing string superprimitivity in parallel, *Information Processing Letters* 49, 5 (1994), 235-241.
9. T. Crawford, C.S. Iliopoulos and R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology*, 11 (1998), 73-100.

10. M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Information Processing Letters* 12, 5 (1981), 244-250.
11. M. Crochemore, C.S. Iliopoulos and M. Korda, Two-dimensional Prefix String Matching and Covering on Square Matrices, *Algorithmica* 20 (1998), 353-373.
12. M. Crochemore, C.S. Iliopoulos and H. Yu, Algorithms for computing evolutionary chains in molecular and musical sequences, *Proc. 9th Australasian Workshop on Combinatorial Algorithms* (1998), 172-185.
13. C.S. Iliopoulos and M. Korda, Optimal parallel superprimitivity testing on square arrays, *Parallel Processing Letters* 6, 3 (1996), 299-308.
14. C.S. Iliopoulos, D.W.G. Moore and K. Park, Covering a string, *Algorithmica* 16 (1996), 288-297.
15. C.S. Iliopoulos and L. Mouchard, An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings, to appear in the *Proceedings of CATS'99: "Computing: Australasian Theory Symposium"*, Auckland, New Zealand, Lecture Notes in Computer Science (1999), 262–272.
16. C.S. Iliopoulos and K. Park, A work-time optimal algorithm for computing all string covers, *Theoretical Computer Science* 164 (1996), 299-310.
17. C.S. Iliopoulos and 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 and W.F. Smyth, On-line algorithms for k-covering, *Proc. 9th Australasian Workshop on Combinatorial Algorithms* (1998), 97-106.
19. D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM Journal on Computing* 6, 1 (1977), 323-350.
20. G.M. Landau, E.W. Myers and J.P. Schmidt, Incremental string comparison, *SIAM Journal on Computing* 27, 2 (1998), 557-582.
21. G.M. Landau and J.P. Schmidt, An algorithm for approximate tandem repeats, *Proc. 4th Symp. Combinatorial Pattern Matching*, Lecture Notes in Computer Science 648 (1993), 120-133.
22. Y. Li and W.F. Smyth, An optimal on-line algorithm to compute all the covers of a string, preprint.
23. D. Maier, The complexity of some problems on subsequences and supersequences, *J. Assoc. Comput. Mach.* 25 (1978), 322-336.
24. M.G. Main and R.J. Lorentz, An algorithm for finding all repetitions in a string, *Journal of Algorithms* 5 (1984), 422-432.
25. M. Middendorf, More on the complexity of common superstring and supersequence problems, *Theoretical Computer Science* 125, 2 (1994), 205-228.
26. 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 (1995), 101-103.
27. K.J. Räihä and E. Ukkonen, The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science* 16 (1981), 187-198.
28. 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 (1998), 972-992.
29. P.H. Sellers, Pattern recognition genetic sequences by mismatch density, *Bulletin of Mathematical Biology* 46, 4 (1984), 501-514.
30. L. Wang and T. Jiang, On the complexity of multiple sequence alignment, *J. Comp. Biol.* 1 (1994), 337-348.