Finding all covers of an indeterminate string in $O(n)$ time on average

Md. Faizul Bari, M. Sohel Rahman, and Rifat Shahriyar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
{faizulbari, msrahman, rifat}@cse.buet.ac.bd

Abstract. We study the problem of finding all the covers of an indeterminate string. An indeterminate string is a sequence $T = T[1]T[2]\cdots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. Here we describe an algorithm for finding all the covers of a string x. The algorithm is applicable for both regular and indeterminate strings. Our algorithm starts with the border array and uses pattern matching technique of the Aho-Corasick Automaton to compute all the covers of x from the border array. On average the algorithm requires $O(n)$ time to find out all the covers, where n is the length of x. Finally, we extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity.

Keywords: indeterminate strings, covers, cover array, Aho-Corasick Automaton, string regularities

1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology, repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory, regularities are important in the analysis of stochastic processes. In computer science, repetitive elements in strings are important in e.g. data compression, computational music analysis, coding, automata and formal language theory. As a result, in the last 20 years, string regularities have drawn a lot of attention from different disciplines of science.

The most common forms of string regularities are periods and repeats and there are several $O(n \log n)$ time algorithms for finding repetitions [6,8], in a string x, where n is the length of x. Apostolico and Breslauer [4] gave an optimal $O(\log \log n)$-time parallel algorithm for finding all the repetitions of a string of length n. The preprocessing of the Knuth-Morris-Pratt algorithm [14] finds all periods of every prefix of x in linear time.

In many cases, it is desirable to relax the meaning of repetition. For instance, if we allow overlapping and concatenations of periods in a string we get the notion of *covers*. After periods and repeats, cover is the most popular form of regularities in strings. The idea of cover generalizes the idea of periods or repeats. A substring c of a string x is called a *cover* of x if and only if x can be constructed by concatenation and superposition of c. Another common string regularity is the *seed* of a string. A seed is an extended cover in the sense that it is a cover of a superstring of x.

Clearly, x is always a cover of itself. If a proper substring pc of x is also a cover of x, then pc is called a *proper cover* of x. For example, the string $x = abcababcabcabcab$ has covers x and $abcab$. Here, $abcab$ is a proper cover. A string that has a proper cover is called *coverable*; otherwise it is *superprimitive*. The notion of covers was introduced

Md. Faizul Bari, M. Sohel Rahman, Rifat Shahriyar: Finding all covers of an indeterminate string in $O(n)$ time on average, pp. 263–271.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

by Apostolico, Farach and Iliopoulos in [5], where a linear-time algorithm to test the superprimitivity of a string was given. Moore and Smyth in [16] gave linear time algorithms for finding all covers of a string.

In this paper, we are interested in regularities of indeterminate strings. An indeterminate string is a sequence $T = T[1]T[2]\cdots T[n]$, where $T[i] \subseteq \Sigma$ for $1 \leq i \leq n$, and Σ is a given fixed alphabet. If $|T[i]| > 1$, then the position i of T is referred to as an *indeterminate position*. The simplest form of indeterminate string is one in which each indeterminate position can only contain a don't care character, denoted by '*'; the don't care character matches any letter in the alphabet Σ. Effectively, $* = \{\sigma_i \in \Sigma \mid 1 \leq i \leq |\Sigma|\}$. The pattern matching problem with don't care characters has been solved by Fischer and Paterson [9] more than 30 years ago. However, although the algorithm in [9] is efficient in theory, it is not very useful in practice. Pattern matching problem for indeterminate string has also been investigated in the literature, albeit with little success. For example, in [13], an algorithm was presented which works well only if the alphabet size is small. Pattern matching for indeterminate strings has mainly been handled by bit mapping techniques (Shift-Or method) [7,18]. These techniques have been used to find matches for an indeterminate pattern p in a string x [11] and the `agrep` utility [17] has been virtually one of the few practical algorithms available for indeterminate pattern-matching.

In [11] the authors extended the notion of indeterminate string matching by distinguishing two distinct forms of indeterminate match, namely, *quantum* and *deterministic*. Roughly speaking, a quantum match allows an indeterminate letter to match two or more distinct letters during a single matching process; a determinate match restricts each indeterminate letter to a single match [11].

Very recently, the issue of regularities in indeterminate string has received some attention. For example, in [2], the authors investigated the regularities of *conservative* indeterminate strings. In a conservative indeterminate string the number indeterminate positions is bounded by a constant. The authors in [2] presented efficient algorithms for finding the smallest *conservative cover* (number of indeterminate position in the cover is bounded by a given constant), λ-conservative covers (conservative covers having length λ) and λ-conservative seeds. On the other hand, Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of an indeterminate string in [3] and showed that their algorithm can be easily extended to compute all the covers of x. The later algorithm runs in $O(n^2 \log n)$ time.

In this paper, we devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case. We also show that our algorithm works in $O(n)$ time on the average. We also extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case. Notably, our algorithm, unlike the algorithm of [2], does not enforce the restriction that the cover or the input string x must be conservative.

The rest of this paper is organized as follows. Section 2 gives account of definitions and notations. Section 3 presents our algorithm to find out all the covers of x. In Section 4, we extend our algorithm to compute the cover array. Finally, Section 5 gives the conclusions.

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 $w \in \Sigma$, is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i \dots j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string $w \in \Sigma$ is a *prefix* (*suffix*) of x if $x = wy$ ($x = yw$), for $y \in \Sigma^*$. A string w is a *subsequence* of x (or x 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 *abcabbcde*. For a given set S of strings, a string w is called a common subsequence of S if s is a subsequence of every string in S.

The string xy is the *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1 \dots n]$ and $y = y[1 \dots m]$ such that $x[n-i+1 \dots n] = y[1 \dots i]$ for some $i \geq 1$ (i.e., x has a suffix equal to a prefix of y), the string $x[1 \dots n]y[i+1 \dots m]$ is said to be a *superposition* of x and y. We also say that x *overlaps* with y. A substring y of x is called a *repetition* in x, if $x = uy^k v$, 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 non-empty substring w is called a *period* of a string x, if x can be written as $x = w^k w'$ 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*. The *cover array* γ, is a data structure used to store the length of the longest proper cover of every prefix of x; that is for all $i \in \{1, \dots, n\}$, $\gamma[i]$ = length of the longest proper cover of $x[1 \dots i]$ or 0. In fact, since every cover of any cover of x is also a cover of x, it turns out that, the cover array γ describes all the covers of every prefix of x. A substring w of x is called a *seed* of x, if w covers a superstring of x[1]. For example, *aba* and *ababa* are some seeds of $x = ababaab$. A border u of x is a prefix of x that is also a suffix of x; thus $u = x[1 \dots b] = x[n-b+1 \dots n]$ for some $b \in \{0, \dots, n-1\}$. The border array of x is an array β such that for all $i \in \{1, \dots, n\}$, $\beta[i]$ = length of the longest border of $x[1 \dots i]$. Since every border of any border of x is also a border of x, β encodes all the borders of every prefix of x. Depending on the matching of letters, borders of indeterminate strings can be of two types, namely, the *quantum border* and the *deterministic border*. Roughly speaking, a quantum match allows an indeterminate letter to match two or more distinct letters during a single matching process, whereas, a determinate match restricts each indeterminate letter to a single match. The notion of these two type of borders was introduced in [10].

An indeterminate string is a sequence $T = T[1]T[2] \cdots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. When a position of the string is indeterminate, and it can match more than one element from the alphabet Σ, we say that this position has non-solid symbol. If in a position we have only one element of the alphabet Σ present, then we refer to this symbol as solid. In an indeterminate string a non-solid position can contain up to $|\Sigma|$ symbols. So, to check whether the two positions match in the traditional way, we would need $|\Sigma|^2$ time in the worst

[1] Note that, x is a superstring of itself. Therefore, every cover is also a seed but the reverse is not necessarily true.

case. To perform this task efficiently we use the following idea originally borrowed from [15] and later used in [3,2]. We use a bit vector of length $|\Sigma|$ as follows:

$$\forall T \subseteq \Sigma, \qquad \nu[T] = [\nu_{t_1}, \nu_{t_2}, \ldots, \nu_{t_{|\Sigma|}}],$$
$$\text{where } \forall t_i \in \Sigma \qquad \nu_{t_i} = \begin{cases} 1, \text{ if } t_i \in T \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

During the string matching process instead of comparing $T[i]$ with $T[j]$, we test bit vectors $\nu[T[i]]$ and $\nu[T[j]]$ using bit operation *AND*. Thus we can compare any two symbols in constant time since the alphabet size is fixed.

In our algorithm we heavily use the Aho-Corasick Automaton invented by Aho and Corasick in [1]. The Aho-Corasick Automaton for a given finite set P of patterns is a Deterministic Finite Automaton G accepting the sets of all words containing a word of P as a suffix. More formally, $G = (Q, \Sigma, g, f, q_0, F)$, where function Q is the set of states, Σ is the alphabet, g is the forward transition, f is the failure link i.e. $f(q_i) = q_j$, if and only if S_j is the longest suffix of S_i that is also a prefix of any pattern, q_0 is the initial state and F is the set of final (terminal) states [1]. The construction of the AC automaton can be done in $O(d)$-time and space complexity, where d is the size of the dictionary, i.e. the sum of the lengths of the patterns which the AC automata will match.

3 Our Algorithm

We start with a formal definition of the problem we handle in this paper.

Problem 1. Computing All Covers of an Indeterminate String over a fixed alphabet.
Input: We are given an indeterminate string x, of length n on a fixed alphabet Σ.
Output: We need to compute all the covers of x.

Our algorithm depends on the following facts:

Fact 1. *Every cover of string x is also a border of x.*

Fact 2. *If u and c are covers of x and $|u| < |c|$ then u must be a cover of c.*

The running time analysis of our algorithm depends on Lemma 3 which is a extension of the analysis provided in [12] by Iliopoulos et al. where they have showed that number of borders of a regular string with a don't care is bounded by 3.5 on average. Here we have extended it for indeterminate strings and proved that, the expected number of borders of an indeterminate string is also bounded by a constant.

Lemma 3. *The expected number of borders of an indeterminate string is bounded by a constant.*

Proof (Proof of Lemma 3). We suppose that the alphabet Σ consists of ordinary letters $1, 2, \ldots, \alpha$, $\alpha \geq 2$. First we consider the probability of two symbols of a string being equal. Equality occurs in the following cases:

Symbol	Match To	Number of cases
$\sigma \in \{1, 2, \ldots, \alpha\}$	$\sigma \in \{1, 2, \ldots, \alpha\}$	α
$\sigma \in S,\ S \subseteq \Sigma$	$\sigma \in S,\ S \subseteq \Sigma,\ \lvert S\rvert > 1$	$\sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$

Thus the total number of equality cases is $\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$ and the number of overall cases is $2^{2\alpha}$. Therefore the probability of two symbols of a string being equal is

$$\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}$$

Now let's consider the probability of string x having a border of length k. If we let $P[expression]$ denotes the probability that the *expression* hold, then

$$\begin{aligned} P\Big[x[1 \ldots k] = x[n-k+1 \ldots n]\Big] &= P\Big[x[1] = x[n-k+1]\Big] \times \cdots \times P\Big[x[k] = x[n]\Big] \\ &= \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k \end{aligned}$$

From this it follows that the expected number of borders is

$$\sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k$$

This summation assumes its maximum value when α is equal to 12 and the summation is bounded above by

$$\begin{aligned} \sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k &\leq \left(\frac{16221211}{16777216}\right) + \left(\frac{16221211}{16777216}\right)^2 + \cdots \\ &\quad + \left(\frac{16221211}{16777216}\right)^{n-1} \\ &= \frac{1 - \left(\frac{16221211}{16777216}\right)^n}{1 - \left(\frac{16221211}{16777216}\right)} - 1 \\ &= \frac{1}{\frac{556005}{16777216}} - 1 \\ &= 29.1746 \end{aligned}$$

So, the expected number of borders of an indeterminate string is bounded by 29.1746. □

Now by using Fact 1, we can compute all the covers of x from its border array. This can be done simply by checking each border and finding out whether it covers x or not. Our algorithm is based on this approach. Broadly speaking, our algorithm consists of two steps. In the first step, the deterministic border array of x is computed.

For this we have used the algorithm introduced by Holub and Smyth in [10], that can compute the deterministic border array of an indeterminate string x in expected $O(n)$ time and space complexity, where n is the length of x. In the second step, we check each border whether it is a cover of x or not. Utilizing Fact 2 and the pattern matching technique of Aho-Corasick Automaton, this step can be performed in $O(n)$ time and space complexity on average. In what follows, we explain the steps of the algorithm in more details.

3.1 First Step: Computing the Border Array

In the first step, we utilize the algorithm provided by Holub and Smyth [10] for computing the deterministic border of an indeterminate string. The output of the algorithm is a two dimensional list β. Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment and the list is kept sorted in decreasing order of border lengths of $x[1 \dots i]$. So the first entry of β_i represents the largest border of $x[1 \dots i]$. The algorithm is given below just for completeness.

Algorithm 1 Computing deterministic border array of string x

```
1: β_i[j] ← φ,          ∀i, j ∈ {1..n}
2: ν_i ← ν[x[i]],       ∀i ∈ {1..n}      {set bit vector for each x[i]}
3: for i ← 1 to n − 1 do
4:    for all b, β_i[b] ≠ φ do
5:       if 2b − i + 1 < 0 then
6:          p ← ν_{b+1} AND ν_{i+1}
7:       else
8:          p ← β_i[b + 1] AND ν_{i+1}
9:       end if
10:      if p ≠ 0^{|Σ|} then
11:         β_{i+1}[b + 1] ← p
12:      end if
13:   end for
14:   p ← ν_1 AND ν_{i+1}
15:   if p ≠ 0^{|Σ|} then
16:      β_{i+1}[1] ← p
17:   end if
18: end for
```

If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$. But from Lemma 3 we know that the expected number of borders of an indeterminate string is bounded by a constant. As a result the expected running time of the above algorithm is $O(n)$.

3.2 Second Step: Checking the Border for a Cover

In the second step, we find out the covers of string x. Here we need only the last entry of the border array, β_n, where $n = |x|$. If $\beta_n = \{b_1, b_2, b_3\}$ then we can say that x has three borders, namely $x[1 \dots b_1]$, $x[1 \dots b_2]$ and $x[1 \dots b_3]$ of length b_1, b_2 and b_3 respectively and $b_1 > b_2 > b_3$. If the number of borders of x is m then number of entry in β_n is m. We iterate over the entries of β_n and check each border in turn to find out whether it covers x or not. To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton. Simple speaking, we build an Aho-Corasick automaton with the dictionary containing the border of x and

parse x through the automaton to find out whether x can be covered by it or not. Suppose in iteration i, we have the length of the ith border of β_n equal to b. In this iteration, we build an Aho-Corasick automaton for the following dictionary:

$$D = \{x[1]x[2]\cdots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b }, \ x[j] \in \Sigma \tag{2}$$

Then we parse the input string x through the automaton to find out the positions where the pattern $c = x[1 \ldots b]$ occurs in x. We store the starting index of the occurrences of c in x in a position array P of size $n = |x|$. We initialize P with all entries set to zero. If c occurs at index i of x then we set $P[i] = 1$. Now if the distance between any two consecutive 1's is greater than the length of the border b then the border fails to cover x, otherwise c is identified as a cover of x. We store the length of the covers in an array AC. At the end of the process AC contains the length of all the covers of x. The definition of AC can be given as follows:

$$AC = \{c_1, c_2, \ldots, c_k\}, \qquad \text{where } \forall i \in 1 \text{ to k}, \ c_i \text{ is a cover of } x \tag{3}$$

Algorithm 2 formally presents the steps of a function $isCover()$, which is the heart of Step 2 described above.

Algorithm 2 Function isCover(x, c)

1: Construct the Aho-Corasick automaton for c
2: parse x and compute the positions where c occurs in x and put the positions in the array Pos
3: **for** $i = 2$ to $|Pos|$ **do**
4: **if** $Pos[i] - Pos[i-1] > |c|$ **then**
5: Return FALSE
6: **end if**
7: **end for**
8: Return TRUE

The time and space complexity of Algorithm 2 can be deduced as follows. Clearly, Steps 3 and 2 run in $O(n)$. Now, the complexity of Step 1 is linear in the size of the dictionary on which the automaton is build. Here the length of the string in the dictionary can be $n - 1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

A further improvement in running time is achieved as follows. According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c. Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \cdots > b_n$. Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in \{i+1, \ldots, m\}$, if b_j is a cover of x if and only if b_j is a cover of b_i. So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$ this policy improves the overall running time of the algorithm. Algorithm 3 formally present the overall process of Step 2 described above.

Algorithm 3 Computing all covers of x

```
1: k ← n
2: AC ← φ {AC is a list used to store the covers of x}
3: for all b ϵ β_n do
4:     if isCover(x[1..k], x[1..b]) = true then
5:         m ← b
6:         AC.Add(k)
7:     end if
8: end for
```

The running time of Algorithm 3 is $O(nm)$, where m is number of borders of x or alternatively number of entries in β_n. Again, from Lemma 3 we can say that the number of borders of an indeterminate string is bounded by a constant on average. Hence, the expected running time of Algorithm 3 is $O(n)$.

It follows from above that our algorithm for finding all the covers of an indeterminate string of length n runs in $O(n)$ time on the average. The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$, which is also an improvement since the current best known algorithm [3] for finding all covers requires $O(n^2 \log n)$ in the worst case.

4 Computing Cover Array

The cover array γ, is to store the length of the longest proper cover of every prefix of x; that is for all $i \in \{1, \ldots, n\}$, $\gamma[i]$ = length of the longest proper cover of $x[1 \ldots i]$ or 0. Our algorithm can readily be extended to compute the cover array of x. Algorithm 3 can be used here after some modification to compute the cover array of x. Here we only need the length of the largest border of each prefix of x. This information is stored in the first entry of each β_i of the border array. If the border array is implemented using any traditional two dimensional list even then the first entry of each list can be accessed in constant time. Let us assume that $\beta_i[1]$ denotes the first entry of the list β_i that is $\beta_i[1]$ is the length of the largest border of $x[1 \ldots i]$.

Algorithm 4 Computing cover array γ of x

```
1: γ[i] ← 0      ∀ i ϵ {1...n}
2: for i ← 1 to n do
3:     if isCover(x, x[1...β_i[1]]) = true then
4:         γ[i] ← β_i[1]
5:     end if
6: end for
```

The $isCover(x, c)$ function (Algorithm 2) is used to identify the covers of x. The algorithm iterates over the first entries of the border array β and checks the borders one by one. If the border $x[1 \ldots \beta_i[1]]$ is identified as a cover then $\gamma[i]$ is set equal to $\beta_i[1]$. otherwise its set to zero. The steps are formally presented in Algorithm 4.

As the worst case running time of the $isCover(x, c)$ function is $O(n)$ and the algorithm iterates over the n lists of the border array β, the running time of Algorithm 4 is $O(n^2)$. The space requirement to store the cover array γ is clearly linear in the length of x, so the space complexity is $O(n)$.

5 Conclusions

In this paper we have presented an average case $O(n)$ time and space complex algorithm for computing all the covers of a given indeterminate string x of length n. We

have also presented an algorithm for computing the cover array γ of an indeterminate string. This algorithm requires $O(n^2)$ time and $O(n)$ space in the worst case. Both of these algorithms are improvement over existing algorithms.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau: *Conservative string covering of indeterminate strings.* Proceedings of the Prague Stringology Conference 2008, 2008, pp. 108–115.
3. P. Antoniou, C. S. Iliopoulos, I. Jayasekera, and W. Rytter: *Computing repetitive structures in indeterminate strings.* Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008), 2008.
4. A. Apostolico and D. Breslauer: *An optimal $O(\log \log n)$-time parallel algorithm for detecting all squares in a string.* SIAM Journal of Computing, 25(6) 1996, pp. 1318–1331.
5. A. Apostolico, M. Farach, and C. S. Iliopoulos: *Optimal superprimitivity testing for strings.* Information Processing Letters, 39(11) 1991, pp. 17–20.
6. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string.* Theory of Computer Science, 22 1983, pp. 297–315.
7. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
8. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Information Processing Letters, 12(5) 1981, pp. 244–250.
9. M. J. Fischer and M. S. Paterson: *String-matching and other products.* Technical report, Cambridge, MA, USA, 1974.
10. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings.* Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, 2003, pp. 36–45.
11. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50.
12. C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. G. Perdikuri, W. F. Smyth, and A. K. Tsakalidis: *String regularities with don't cares.* Nordic Journal of Computing, 10(1) 2003, pp. 40–51.
13. C. S. Iliopoulos, L. Mouchard, and M. S. Rahman: *A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching.* Mathematics in Computer Science, 1(4) 2008, pp. 557–569.
14. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
15. I. Lee, A. Apostolico, C. S. Iliopoulos, and K. Park: *Finding approximate occurrences of a pattern that contains gaps*, in Proceedings 14-th Australasian Workshop on Combinatorial Algorithms, SNU Press, 2003, pp. 89–100.
16. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string.* Information Processing Letters, 50(5) 1994, pp. 239–246.
17. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool.* In Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162.
18. S. Wu and U. Manber: *Fast text searching: allowing errors.* Communications of the ACM, 35(10) 1992, pp. 83–91.