String Regularities with Don't Cares

Costas S. Iliopoulos[1†], Manal Mohamed[1‡], Laurent Mouchard[2], Katerina G. Perdikuri[3], W. F. Smyth[4] and Athanasios K. Tsakalidis[3]

[1] Department of Computer Science, King's College London, London WC2R 2LS, England
{csi,manal}@dcs.kcl.ac.uk
[2] Department of Vegetal Physiology - ABISS, Université de Rouen, 76821 Mont Saint Aignan Cedex, France
Laurent.Mouchard@univ-rouen.fr
[3] Computer Technology Institute, Patras, Greece
perdikur@ceid.upatras.gr, tsak@cti.gr
[4] Algorithms Research Group, Department of Computing & Software, McMaster University, Hamilton, Ontario, Canada L8S 4K1 and School of Computing, Curtin University, Perth WA 6845, Australia
smyth@mcmaster.ca

Abstract. We describe algorithms for computing typical regularities in strings $x = x[1..n]$ that contain don't care symbols. For such strings on alphabet Σ, an $O(n \log n \log |\Sigma|)$ worst-case time algorithm for computing the period is known, but the algorithm is impractical due to a large constant of proportionality. We present instead two simple practical algorithms that compute all the periods of every prefix of x; our algorithms require quadratic worst-case time but only linear time in the average case. We then show how our algorithms can be used to compute other string regularities, specifically the covers of both ordinary and circular strings.

Key words: string algorithm, regularities, don't care, period, border, cover.

1 Introduction

Regularities in strings arise in many areas of science: combinatorics, coding and automata theory, molecular biology, formal language theory, system theory, etc. — they thus form the subject of extensive mathematical studies (see e.g. [L83],[P93],[P90]). Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns. A typical regularity, the period u of the string x, grasps the repetitiveness of x, since x is a prefix of a string constructed by

† Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.
‡ Supported by EPSRC studentship.

concatenations of u. Here we consider regularity problems that arise from having "don't care" symbols in the string. In particular we study string problems focused on finding the repetitive structures in DNA strings x.

In this paper we also consider a kind of generalized period called a *cover*; that is, a proper substring u of x (if it exists) such that x can be formed by concatenating and overlapping occurrences of u. In the computation of covers, two main problems have been considered in the literature: the *shortest-cover* problem (computing the shortest cover of a given string of length n), and the *all-covers* problem (computing all the covers of a given string). Apostolico, Farach and Iliopoulos [AFI91] introduced the notion of covers and gave a linear-time algorithm for the shortest-cover problem. Breslauer [B92] presented a linear-time on-line algorithm for the same problem. Moore and Smyth [MS95] presented a linear-time algorithm for the all-covers problem. Finally, Li and Smyth [LS02] invented the cover array and described an on-line linear-time algorithm that solves both the shortest-cover and all-covers problems for every prefix of x. In parallel computation, Breslauer [B94] gave an optimal $O(\alpha(n)\log\log n)$-time algorithm for the shortest cover, where $\alpha(n)$ is the inverse Ackermann function; Iliopoulos and Park [IP94] gave an optimal $O(\log\log n)$-time (thus work-time optimal) algorithm for the same problem.

The idea of a cover has been extended. Iliopoulos, Moore and Park [IMP96] introduced the notion of seeds and gave an $O(n\log n)$-time algorithm for computing all the seeds of a given string of length n. For the same problem Ben-Amram, Berkman, Iliopoulos and Park [BBIP94] presented a parallel algorithm that requires $O(\log n)$ time and $O(n\log n)$ work. Apostolico and Ehrenfeucht [AE93] considered yet another problem related to covers.

An interesting extension of string-matching problems with practical applications in the area of DNA sequences results from the introduction of "don't care" symbols. A *don't care* symbol $*$ has the property of matching with any symbol in the given alphabet. For example the string $p = AC*C*$ matches the pattern $q = A*DCT$. Exact string matching with "don't care" symbols was studied by Fischer and Paterson [FP74]. They developed an $O(n\log m\log|\Sigma|)$ time algorithm for finding a pattern of length m in a text of size n over the alphabet $\Sigma\cup\{*\}$. Their method is based on the theoretically fast computation method of convolutions, but it is not efficient in practice. Pinter developed a linear time algorithm for a special case [P85], while Abrahamson generalized Fischer and Paterson's algorithm, using a divide-and-conquer approach that runs in time $O(n\sqrt{m\log m})$ [A87]. See also [LV89].

In this paper we describe two fast, practical algorithms for computing all the periods of every prefix of a given string $x[1..n]$ that contains "don't care" symbols. We prove that the expected running time of these algorithms is linear, though they have quadratic worst-case time complexity for pathological inputs. Then we show how our algorithms can be used to efficiently compute covers of strings with don't cares, both ordinary and circular. The motivation for the above problems comes from many applications to the analysis of DNA sequences that reveal naturally occurring repeated segments within nucleotide sequences. These segments can be concatenated only (periodic) or both concatenated and overlapping (coverable).

2 Background

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all nonempty strings over the alphabet Σ is denoted by Σ^+. A string x of length n is represented by $x[1..n] = x[1]x[2]\cdots x[n]$, where $x[i] \in \Sigma$ for $1 \leq i \leq n$, and $n = |x|$ is the length of x. The empty string is the empty sequence (of zero length) and is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k and is called *the* k^{th} *power of* x.

A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$, a *proper prefix* if $u \in \Sigma^+$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$. A string u that is both a proper prefix and a suffix of x is called *a border* of x.

If x has a nonempty border, it is called *periodic*. Otherwise, x is is said to be *primitive*. The empty string is a trivial border of x. Let u denote a border of x of length ℓ where $1 \leq \ell \leq n-1$; then $p = n - \ell$ is called *a period* of x. Clearly, p is a period of x if $x_i = x_{i+p}$ whenever $1 \leq i, i+p \leq n$. Another equivalent definition may be given as: p is a period of x if and only if $x[1..p] = x[n-p+1..n]$. The latter definition shows that each word x has a minimum period called *the period* of x. For example, the string $x = ababab$ has two borders $u_1 = ab$ and $u_2 = abab$; thus x has two periods 4 and 2, where 2 is *the* period of x.

A substring u is said to be a *cover* of a given string x if every position of x lies within an occurrence of a string u within x. If, in addition, $|u| < |x|$, we call u a *proper cover* of x. For example, x is always a cover of x. and $u = aba$ is a proper cover of $x = abaababa$.

An array $\beta[1..n]$ is called the *border array* of $x[1..n]$, where for $i = 1, 2, \ldots, n$, $\beta[i]$ gives the length of the longest border of $x[1..i]$. Furthermore, since every border of a border of x is itself a border of x, β actually describes all the borders of every prefix of x. The border array can be computed in linear time using the classical failure function algorithm [AHU74].

Recently Li and Smyth [LS02] discovered the cover array $\gamma[1..n]$, where $\gamma[i]$ gives the length of the longest cover of $x[1..i]$. The cover array similarly encapsulates all the covers of every prefix of x and can also be computed in linear time.

This paper deals with strings that can contain occurrences of the *don't care* symbol, denoted by "$*$". This symbol matches any other symbol of the alphabet. Two symbols a and b match ($a \approx b$) if they are equal, or if one of them is a don't care symbol. Notice that the relation $\approx$ is not transitive ($a \approx *, * \approx b \not\Rightarrow a \approx b$).

3 Computing the Failure Function

A theoretical $O(n \log n \log |\Sigma|)$ time algorithm for computing the period of a given string x that contains don't care symbols can be achieved by using a "convolution" procedure [FP74] between two strings x and X. Assuming that x is the given string (of length n), we create a string X by adding n don't care symbols, thus doubling the length of x. We compute the convolution of x and X by shifting x to the right by one character. The product u of the convolution is the period of the string x (for

further information see [FP74]). This algorithm is impractical as it has a very large constant hidden in its asymptotic time complexity.

In this section we present two fast and practical algorithms for computing the border array $\beta[1 \ldots n]$ of a given string x that contains don't care symbols.

As noted earlier, the standard failure function method, based on the fact that "a border of a border of a string x is necessarily a border of x", cannot be used to calculate the border array of a string containing don't care symbols. This follows from the nontransitivity of the $\approx$ relation. For example, if $x = a**ca$, then we have

$$u_l = a** \approx u_r = *ca,$$

where u_l and u_r are respectively the left and right borders of x of length 3; note that $v_l = a* \approx **$ is a border of u_l, but $a* \neq ca$, which means that v_l is not a border of u_r, hence not of x.

Despite the fact that we cannot make use of the standard failure function method, it is quite easy to notice that there is no nonempty border b of $x[1..i+1]$ that is not equal to some $b'x[i+1]$, where b' is a border of $x[1 \ldots i]$. Moreover, let the borders of $x[1..i]$ be

$$\beta^1[i], \beta^2[i] \ldots \beta^k[i]$$

where $\beta^1[i]$ is the the length of the border of $x[1 \ldots i]$ (the longest border) and $\beta^k[i] = 0$ is the length of the empty border .Then each border of $x[1 \ldots i+1]$ is equal to either $\beta^j[i] + 1$ for some $1 \leq j \leq k$ or 0.

The above states the rule used by algorithm FAILURE-FUNCTION-1() to calculate the value of the border array of a given string x that contains don't care symbols.

FAILURE-FUNCTION-1(x)
1 $S \leftarrow \emptyset$ S is a singly-linked list of nonzero border lengths
2 $\beta[1] \leftarrow 0$
3 **For** $i \leftarrow 1$ **To** $n-1$ **Do**
4 **For** each $b \in S$ **Do**
5 **If** $x[i+1] \approx x[b+1]$ **Then**
6 replace_current($S, b+1$)
7 **Else** delete_current(S)
8 **If** $x[i] \approx x[1]$ **Then** add_after_current(S,1)
9 **If** $S \neq \emptyset$ **Then** $\beta[i+1] \leftarrow$top(S)
10 **Else** $\beta[i+1] \leftarrow 0$
END FAILURE-FUNCTION-1

Figure 1: FAILURE-FUNCTION-1 algorithm.

The algorithm maintains a list S of all possible nonzero border lengths. At the beginning of iteration i, S contains all possible nonzero border lengths of $x[\ldots i]$. The algorithm tries to extend each possible border b in S by comparing the value of $x[i+1]$ and the value of $x[b+1]$. If the two values are equal or one of them is a don't care symbol, the value b in S is replaced by $b+1$. Otherwise, b is deleted from the list. If $x[i+1]$ is equal to $x[1]$ or $*$, a border of length 1 has to be added to S. Finally,

each iteration i terminates by assigning the value at the top of the list S that is the length of the longest border of $x[1 \ldots i+1]$ to $\beta[i+1]$. If the list S is empty, then the length of the longest border is 0 ($\beta[i+1] = 0$). Note that at this stage , S contains the lengths of all possible nonzero borders of $x[1 \ldots i+1]$ in descending order.

Each position i such that $x[i] = x[1]$ or $*$ is a candidate to start a new border. Hence Algorithm FAILURE-FUNCTION-2() tries to speed up the computation of the failure function by a simple linear preprocessing of the input string x. For each position i we count the previous occurrences of $x[1]$'s and $*$'s. And we introduce a pointer that points to the previous occurrence. The algorithm then modifies the standard failure function method to calculate the border array β. FAILURE-FUNCTION-2 starts by setting the value of $\beta[0]$ to -1, a convention which is compatible with the algorithm. Then $n-1$ iterations follow. In each iteration i, the algorithm tries to extend the current border b by comparing the value of $x[i+1]$ and the value of $x[b+1]$ where b is the length of the border of $x[1 \ldots i]$. If the two values are equal or one of them is a don't care symbol, the value of $\beta[i]$ is set to $b+1$. Otherwise, the algorithm tries to follow the basic failure function method by trying to extend the border of the current border. More work needs to be done in each attempt to ensure the right answer:

- The algorithm has to eliminate the possibility of having a border whose length is greater than that of the border of the border. That is, having

  $$x[1 \ldots i-j+2] \approx x[j \ldots i+1]$$

  for some j such that $\beta[b] < i-j+1 < b$. The algorithm uses the preprocessed informations to find each position j such that $x[j] = x[1]$ or $*$. Clearly, the number and the positions of the j's can be calculated in constant time. The algorithm examines each j in ascending order to find the first j that satisfies the above condition. If such a j exists, then the iteration ends by assigning $i-j+2$ to $\beta[i+1]$.

- Recall that the nontransitivity of the $\approx$ relation means that the statement "the border of the border is a border" may not be true. Observe that nontransitivity can occur only if a don't care symbol was part of the comparison. Then only in such cases does the algorithm need to recheck the positions that could cause a nontransitivity. That is, if $x[i+1] \approx x[\beta[b]]$, then the algorithm still needs to check all the solid characters in the right border; that have been compared with the don't care symbol during the calculation; against the coresponding characters in the left border. These positions are marked during the calculations and stored in a special stack S. Positions are popped from and pushed onto S depending on the length of the current border.

For example, let $x = a**cabcdabc*abca$ and the value of the border array be as follows:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$x[i]$	a	$*$	$*$	c	a	b	c	d	a	b	c	$*$	a	b	c	a
$\beta[i]$	0	1	2	3	3	2	3	0	1	2	3	4	5	6	7	5

At step 7 ($i = 7$) we had failed to extend the current border after comparing $x[4] =$ 'c' with $x[8] =$ 'd'. At the same time we could not find any j that satisfies the first

```
FAILURE-FUNCTION-2(x)
1  β[0] ← −1
2  For i ← 0 To n − 1 Do
3     b ← β[i]
4     If x(i + 1) ≈ x(b + 1) Then β[i + 1] ← b + 1
5     Else
6        While b ≥ 0 And [x(i + 1) ≠ x(b + 1) Or check_stack_fail()] Do
7              For  each j such that β[b] < i − j + 1 < b And x[j] ≈ x[1] Do
8                 If x[j..i + 1] ≈ x[1..i − j + 2] Then
9                    b ← i − j + 2
10                   Quit The While Loop
11             b ← β[b]
12       β[i + 1] ← b
END FAILURE-FUNCTION-2
```

Figure 2: FAILURE-FUNCTION-2 algorithm.

condition. So we tried to extend the border of the border which equals 3 ($\beta[7] = 3$). Since $x[8] \neq x[4]$, we tried to extend the border of the border of the border which equals 2 ($\beta[3] = 2$). Although $x[8] \approx x[3]$, we still need to check according to the algorithm the value at position 1 with the corresponding value at position 6. Since they are not equal, the value of $\beta[8]$ can not be 3 and so we have to carry on . Note that the value 1 had been inserted into the stack after comparing the '$*$' at position 2 with the 'a' at position 1 at step 1.

At step 15, where $x[16] \neq x[8]$, we had failed again to extend the current border. According to the algorithm we have to eliminate the possibility of having a longer border than the border of the border; that is, finding j that satisfies the first condition. In our example, we found $j = 12$. Note that

$$\beta[b] = 3 < i - j + 1 = 15 - 12 + 1 = 4 < b = 7$$

and $x[12] = *$. After finding j we need to compare $x[12 \ldots 16]$ with $x[1 \ldots 5]$. Since they are equal the value of $\beta[16]$ becomes 5.

4 Expected Running Time Analysis

Here we will show that the expected number of borders of a string is bounded by a constant. We suppose that the alphabet Σ consists of ordinary letters $1 \ldots \sigma - 1$ together with the don't care symbol $*$. First we consider the probability of two symbols of a string being equal. Equality occurs in the following cases:

Symbol	Equal to	Number of cases
$*$	$\sigma \in \{1, \ldots, \alpha - 1\}$	$\alpha - 1$
$\sigma \in \{*, 1, \ldots, \alpha - 1\}$	$*$	α
$\sigma \in \{1, \ldots, \alpha - 1\}$	$\sigma \in \{1, \ldots, \alpha - 1\}$	$\alpha - 1$

Thus the total number of equality cases is $3\alpha - 2$ and the number of overall cases is α^2. Therefore the probability of two symbols of a string being equal is

$$\frac{3\alpha - 2}{\alpha^2}$$

Now let consider the probability of string x having a border of length k. One can see

$$P[x_1 \dots x_k = x_{n-k-1} \dots x_n] = P[x_1 = x_{n-k-1}] \dots P[x_k = x_n] = \left(\frac{3\alpha - 2}{\alpha^2}\right)^k$$

From this it follows that the expected number of borders is

$$\Sigma_{k=1}^{n-1} \left(\frac{3\alpha - 2}{\alpha^2}\right)^k < 3.5$$

The algorithm, at iteration i, performs k_i steps, where k_i is the number of the borders of $x[1..i]$. Thus the overall expected time complexity is

$$\Sigma_{k=1}^{n-1} k_i.$$

Since the expected value of each k_i is bounded by 3.5, therefore the expected time of the two border algorithms is O(n).

5 Experimental Results

Using random strings over various alphabet sizes (with the $*$ symbol treated as an additional random letter), we ran FAILURE-FUNCTION-1() and FAILURE-FUNCTION-2(). The running time was calculated for each execution. We used a SUN Ultra Enterprise 300MHz running Solaris Unix. The reported times are the calculation time in seconds, measured by calling the a clock() routine (Figures 3 and 4).

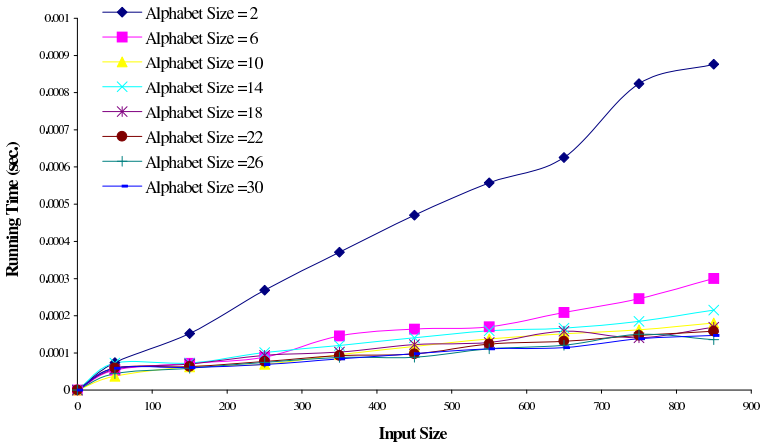


Figure 3: Timing curves for the FAILURE-FUNCTION-1 Procedure.

In general, it seems that the heuristic employed in FAILURE-FUNCTION-2 is effective for random strings on small alphabets (therefore containing a high proportion

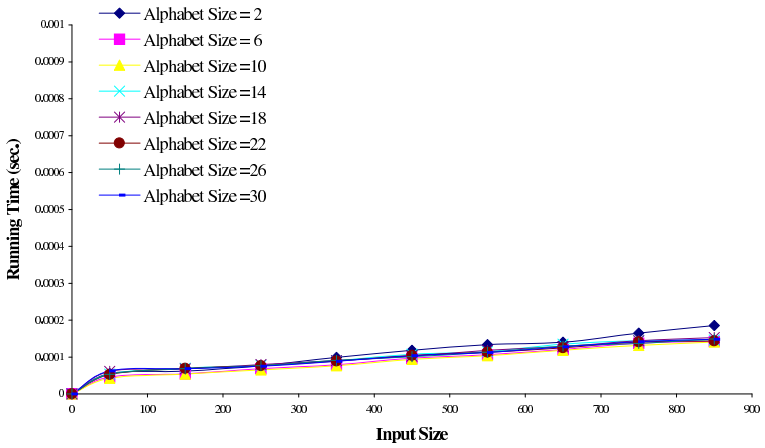


Figure 4: Timing curves for the FAILURE-FUNCTION-2 Procedure.

FIND-COVERS(x)
1 Compute borders $B = \{b_1, \ldots, b_k\}$ of x in ascending order of length
2 **For** each adjacent pair of borders, b_i and b_{i+1}, **Do**
3 **If** b_i covers b_{i+1} **Then** check whether it covers x
4 **Else** $i \leftarrow i + 1$
END FIND-COVERS

Figure 5: FIND-COVERS algorithm.

of don't care symbols), but makes little difference for larger alphabets that have a correspondingly low proportion of don't cares.

Note that our experiments confirm Section 4's theoretical result that the expected case bahaviour of the algorithms is linear in string length.

6 Computing the Covers

In this section we present an algorithm for computing all the covers of a given string x, bearing in mind that we allow possible overlaps. This means that in the example $p = AC*ACA*AA*ACA$, the pattern $q = ACA$ is an overlapping cover of the string p. The algorithm we present consists of 2 stages. The first stage is a preprocessing phase where we compute the borders of the given string x. Suppose we find the following nonempty borders $b_1, b_2, \ldots, b_k$, listed in ascending order.

In the second stage we perform the following check: *for two borders b_i and b_{i+1}, if b_i covers b_{i+1} we check whether b_i also covers string x. If not we continue this process for the rest of the adjacent pairs of borders.*

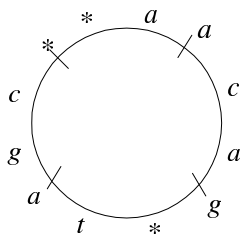
In order to precompute the borders we use Algorithm *ALL-BORDERS()*. Using the previously computed borders, the procedure that finds the covers of a given string x is as follows:

Theorem 6.1 *Given a string x that contains don't care symbols, we can find a longest cover u of x in linear expected time.*

7 Computing the Covers of Circular DNA Strings

In some computational biology applications (for example, DNA sequencing by hybridization), it is convenient to regard the DNA sequence as a circular string (Fig. 6). Given a circular DNA string and a window that limits the region of DNA that we are able to study, the computation of covers in the sequence becomes a difficult task. In that case the computation of seeds (see [BBIP94]) does not work and we need a new approach.

Bearing in mind the scheme of a circular DNA string and the algorithms for the computation of the failure function that we have already described, it is easy to see that the computation of the covers in a circular DNA sequence can be easily solved using the failure function technique. More precisely the problem of the computation of covers can be solved if we compute the failure function two times, once forward and once backward.



S1: a c a g *S2: * t a g* *S3: c * * a*

Figure 6: A circular string x and three substrings S1, S2, S3, as seen from a window of four characters length.

Conclusions

We have presented two linear expected-time algorithms for computing all the borders (hence all the periods) of a given string containing don't care symbols. We have then shown how to apply the border calculation to compute the covers of ordinary and circular strings, also containing don't care symbols.

An open problem is the calculation of every border of every prefix of x in $O(n \log n)$ worst-case time.

References

[A87] Abrahamson, K.: Generalized string matching, SIAM J.Computing, 16, 1039-1051.

[AE93] Apostolico, A., Ehrenfeucht, A.: Efficient detection of quasiperiodicities in strings, Theo. Comp. Sci, 119, 247-265.

[AFI91] Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings, Inform. Process. Lett., 39, 17-20.

[AHU74] Aho, Alfred V., Hopcroft, John E., Ullman, Jeffey D.: The Design & Analysis of Computer Algorithms, Addison-Wesley.

[B92] Breslauer, D.: An on-line string superprimitivity test, Inform. Process. Lett., 44, 345-347.

[B94] Breslauer, D.: Testing string superprimitivity in parallel, Inform. Process. Lett., 49, 235-241.

[BBIP94] Ben-Amram, A.M., Berkman, O.C.S., Iliopoulos, C.S., Park, K.: The subtree max gap problem with application to parallel string covering, Proc. 5th ACM-SIAM Symp. Discrete Algorithms, 501-510.

[FP74] Fischer, M., Paterson, M. : String matching and other products, Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings, 7, 113-125.

[IMP96] Iliopoulos, C.S., Moore, D.W.G., Park, K.: Covering a string, Algorithmica, 16, 288-297.

[IP94] Iliopoulos, C.S., Park, K.: An optimal $O(\log \log n)$ time algorithm for parallel superprimitivity testing, Journal of the Korean Information Science Society, 21-8, 1400-1404.

[L83] Lothaire, M. : Combinatorics on Words, Addison-Wesley, Reading, Mass.

[LS02] Yin Li, Smyth, W.F.: Computing the cover array in linear time, Algorithmica, 32-1, 95-106.

[LV89] Landau, G.M., Viskin, U.: Fast parallel and serial approximate string matching, Journal of Algorithms, 10, 157-169.

[MS95] Moore, D.W.G., Smyth, W.F.: A correction to "Computing the covers of a string in linear time", Inform. Process. Lett., 54,101-103.

[P85] Pinter, R.: Efficient string matching with don't-care patterns, Combinatorial Algorithms on Words, NATO ASI Series, F12, Springer-Verlag, 11-29.

[P90] Pevzner, P.A.: Statistical analysis of genetics texts, Computer Analysis of Genetics Texts, Chapter 2, Ed. M.D. Frank-Kamenetzkii, Nauka, Moscow, 36-80 (in Russian).

[P93] Pevzner, P.A.: Overlapping word paradox and Conway Equation, Supercomputing, Bioinformatics, and Complex Genome Analysis, World Scientific, Ed. C. Cantor, J. Fickett, R. Robbins, and H. Lim, 71-78.