Nordic Journal of Computing 10(2003), 1–12.

STRING REGULARITIES WITH DON'T CARES

COSTAS S. ILIOPOULOS
MANAL MOHAMED
Department of Computer Science, King's College London
London WC2R 2LS, England
{csi|manal}@dcs.kcl.ac.uk

LAURENT MOUCHARD
Department of Vegetal Physiology - ABISS Université de Rouen
76821 Mont Saint Aignan Cedex, France
Laurent.Mouchard@univ-rouen.fr

WILLIAM F. SMYTH*
Algorithms Research Group
Department of Computing & Software
McMaster University, Hamilton
Ontario, Canada L8S 4K1
smyth@mcmaster.ca

KATERINA G. PERDIKURI
ATHANASIOS K. TSAKALIDIS
Research Academic Computer Technology Institute (RACTI)
61 Riga Feraiou str., 26221, Patras, Greece
perdikur@ceid.upatras.gr
tsak@cti.gr

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.

ACM CCS Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.1 [Discrete Mathematics]: Combinatorics – *combinatorial algorithms*

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. Lothaire 1983, Pevzner 1993, Pevzner 1990]. Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns.

*Also: School of Computing, Curtin University, GPO Box U-1987, Perth WA 6845, Australia.

 Received 2003; revised 2003; accepted 2 003.

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 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 *et al.* [1991] introduced the notion of covers and gave a linear-time algorithm for the shortest-cover problem. Breslauer [1992] presented a linear-time on-line algorithm for the same problem. Moore and Smyth [1995] presented a linear-time algorithm for the all-covers problem. Finally, Li and Smyth [2002] 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 [1994] 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 [1994] 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 *et al.* [1996] 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 *et al.* [1994] presented a parallel algorithm that requires $O(\log n)$ time and $O(n \log n)$ work. Apostolico and Ehrenfeucht [1993] 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 [1974]. 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 [1985] developed a linear time algorithm for a special case, while Abrahamson [1987] generalized Fischer and Paterson's algorithm, using a divide-and-conquer approach that runs in time $O(n \sqrt{m \log m})$. See also [Landau and Viskin 1989].

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 $1 \le i \le n$, 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 $1 \le \ell \le n-1$; then $p = n - \ell$ is called *a period* of x. Clearly, p is a period of x if $x_i = x_{i+p}$ whenever $1 \le i, i+p \le n$. 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 [Aho *et al.* 1974].

Recently Li and Smyth [2002] 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.

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 [Fischer and Paterson 1974] 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 Fischer and Paterson 1974]. 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 $\beta[1..n]$ of a given string x that contains don't care symbols.

3.1 Failure-Function-1

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 see that for every border array $\beta[1..n]$ and every integer i such that $1 \leq i < n$, the only possible values of $\beta[i+1]$ are zero or $b+1$, where b denotes any element of the sequence

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

where $\beta^1[i] = \beta[i]$, and $\beta^j[i] = \beta[\beta^{j-1}[i]]$ for every $j \geq 2$ such that $\beta^{j-1}[i]$ is defined and $1 \leq \beta^{j-1}[i] \leq n$. In fact, it is well-known that the above sequence identifies every border of $x[1..n]$. Note that $\beta^k[i] = 0$.

The above states the rule used by FAILURE-FUNCTION-1() (Fig. 1) to calculate the value of the border array $\beta[1..n]$ of a given string x that contains don't care symbols. The algorithm maintains a list L of all possible border lengths in decreasing order. At the beginning of each iteration i, L contains all possible border lengths of $x[1..i]$. That is L, contains all possible values b that are equal to $\beta^j[i]$, $1 \leq j \leq k$.

During each iteration, the algorithm tries to extend all the borders of $x[1..i]$ by comparing the value of $x[i+1]$ with that of $x[b+1]$ for all $b \in L$. If the two values are equal or one of them is a don't care then b in L is replaced by $b+1$. Otherwise, b is deleted from L. Each iteration i is terminated by assigning the first value of the list L to $\beta[i+1]$. Note that at this stage, L contains the lengths of all possible borders of $x[1..i+1]$ in decreasing order.

3.2 Failure-Function-2

Algorithm FAILURE-FUNCTION-2() (Fig. 2) tries to speed up the computation of the failure function by a simple linear preprocessing of the input string x. Each position i such that $x[i] = x[1]$ or $*$ is a candidate to start a border. For each position i, $c_counter[i]$ keeps the number of candidates that appear between positions $1..i$.

```
FAILURE-FUNCTION-1(x)
1  L ← ∅      L is a singly-linked list of all border lengths in ascending order
              each element of the list is an object with a key and pointer fields.
2  β[1] ← 0
3  L ← LIST-INSERT(L, 0)
4  For i ← 1 To n − 1 Do
5     current ← first[L]
6     While current ≠ NIL Do
7        b ← key[current]
8        If x[i + 1] ≈ x[b + 1] Then
9           key[current] ← b + 1
10          current ← next[current]
11       Else
12          temp ← current
13          current ← next[current]
14          LIST-DELETE(L, temp)
15    APPEND(L, 0)
16    β[i + 1] ← key[first[L]]
END FAILURE-FUNCTION-1
```

Fig. 1: FAILURE-FUNCTION-1 algorithm.

We also introduce a pointer that points to the next such candidate. The value of *next_candidat*[i]; where $1 \leq i < n$, keeps the index of the next occurrence of such candidate.

During the computation, the algorithm marks each position j in the left border that contains a solid character (not a don't care), and has been matched to $*$ in the right border. A stack S is used to store such positions. These positions are popped from and pushed onto S depending on the length of the current border. Recall that the relation $\approx$ is not transitive and such non-transitivity occurs only if the don't care symbol was part of the comparison. Then, the border of the border is a border only if the marked positions in the left border match the corresponding positions in the right one.

The algorithm modifies the standard failure function method to calculate the border array $\beta[1..n]$. FAILURE-FUNCTION-2 starts by setting the value of $\beta[1]$ to 0. Then $n-1$ iterations follow. At 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 $x[1..i]$. 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.

Before checking the border of the border, the algorithm has to eliminate the possibility of having a border whose length is greater than that of the border of

```
FAILURE-FUNCTION-2(x)
Initialization
(1)counter ← 0
   For i ← 1 To n
     If x[i] ≈ x[1] Then counter + +
     c_counter[i] ← counter
(2)For i ← 2 To n
     If x[i] ≈ x[1] Then ENQUEUE(Q, i)
   ENQUEUE(Q, n + 1)
   For i ← 1 To n − 1
     If head(Q) = i Then DEQUEUE(Q)
     next_candidate[i] ← head(Q)
(3)Mark[1..n] ← false . . . false
Main Algorithm
1  S ← ∅
2  β[1] ← 0
3  For i ← 1 To n − 1 Do
4    b ← β[i]
5    If EQUAL(b + 1, i + 1, S) Then
6      β[i + 1] ← b + 1
7    Else
8      While (b > 0 And (! EQUAL(b + 1, i + 1, S) Or
                         CHECK-STACK-FAIL(S, b + 1, i + 1))) Do
9          c_no ← c_counter[i − β[b] + 1] − c_counter[i − b + 1] − 1
10         If c_no > 0
11             b′ ← CHECK-CANDIDATES(i − b + 1, i − β[b] + 1, i)
12             If b′ > 0 Then
13                 b ← b′
14                 Quit The While Loop
15         b ← β[b]
16   If x[i + 1] ≈ x[b + 1] Then β[i + 1] ← b + 1
17   Else β[i + 1] ← 0
END  FAILURE-FUNCTION-2
```

Fig. 2: FAILURE-FUNCTION-2 algorithm.

the border. Using the preprocessed information, the algorithm check sequentially all the candidate positions (if there is any) using CHECK-CANDIDATES() subroutine. If the algorithm failed to find a longer border than the border of the border and before accepting the extended border of the border, the algorithm needs to check all the marked positions kept in S with the corresponding positions as mentioned earlier. This is done by using CHECK-STACK-FAIL(). The details of CHECK-CANDIDATES() and CHECK-STACK-FAIL() are provided in Fig. 3.

```
EQUAL(l, r, S)
1 If ((x[l] ≠ ∗ And x[r] = ∗) Or Mark[l]) And S[top[S]] ≠ l Then
2     PUSH(S, l)
3     Mark[l] ← true
4 If x[l] ≈ x[r] Then
5     return true
6 return false

CHECK-CANDIDATES(s, e, p)
1 candidate ← next_candidate[s]
2 While (candidate < e) Do
3     c ← 1
4     While (candidate + c − 1 ≤ p + 1 And x[c] ≈ x[candidate + c − 1])
5         Do c + +
6     If candidate + c − 1 > p + 1 Then
7         return c − 1
8     candidate ← next_candidate[candidate]
8 return 0

CHECK-STACK-FAIL(S, l, r)
1 If top(S) = 0 Then return false
2 While (S[top(S)] ≥ l) Do
3     POP(S)
4 If top(S) = 0 Then return false
5 check ← top(S)        index of the most recently inserted element
6 While (check ≠ 0 And x[S[check]] ≈ x[r − l + S[check]] Do
7     check − −
8 If check = 0 Then return false
9 return true
```

Fig. 3: Subroutines.

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
c_coun	1	2	3	3	4	4	4	4	5	5	5	6	7	7	7	8
next_candi	2	3	5	5	9	9	9	9	12	12	12	13	16	16	16	-
$\beta[i]$	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 candidate that could start a longer border than the border of the border. Accordingly, we tried to extend the border of the border which is equal to 3 ($\beta[7] = 3$). Since $x[8] \neq x[4]$, we again tried to extend the border of the border of the border which is equal to 2 ($\beta[3] = 2$).

Although $x[8] \approx x[3]$, we still need to check the value at the marked 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 pushed onto the stack after comparing the '$*$' at position 2 with 'a' at position 1 (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. Using the preprocessed information, we know that there is only one position that can start a longer border i.e. position 12. Checking such possibility has to be done sequentially comparing the values at $x[12..16]$ with those at $x[1..5]$. Since they are equal the value of $\beta[16]$ is set to 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 $1 \ldots \alpha - 1, \alpha \geq 3$ 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'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

$$\begin{aligned} P[x_1 \ldots x_k = x_{n-k+1} \ldots x_n] &= P[x_1 = x_{n-k+1}] \times \cdots \times P[x_k = x_n] \\ &= \left(\frac{3\alpha - 2}{\alpha^2}\right)^k. \end{aligned}$$

From this it follows that the expected number of borders is

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

This summation assumes its maximum value when α assumes its minimum value 3. Thus the summation is bounded above by

$$\Sigma_{k=1}^{n-1} \left(\frac{3\alpha - 2}{\alpha^2}\right)^k \leq (7/9) + (7/9)^2 + \ldots + (7/9)^{n-1}$$

$$= (1 - (7/9)^n)/(1 - (7/9)) - 1$$
$$= 1/(2/9) - 1$$
$$= 3.5.$$

The algorithm, at iteration i, performs k_i steps, where k_i is the number of 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 both 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 (Fig. 4 and Fig. 5).

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 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 the theoretical result in Section 4 which shows that the expected case behavior of the algorithms is linear in string length.

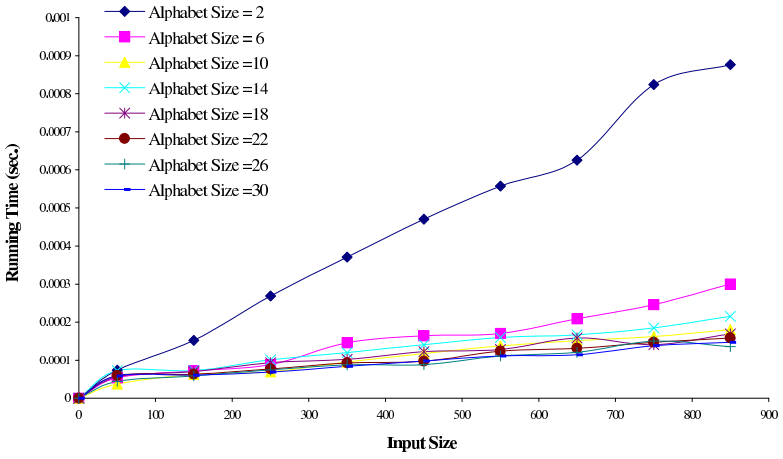


Fig. 4: Timing curves for the FAILURE-FUNCTION-1 procedure.

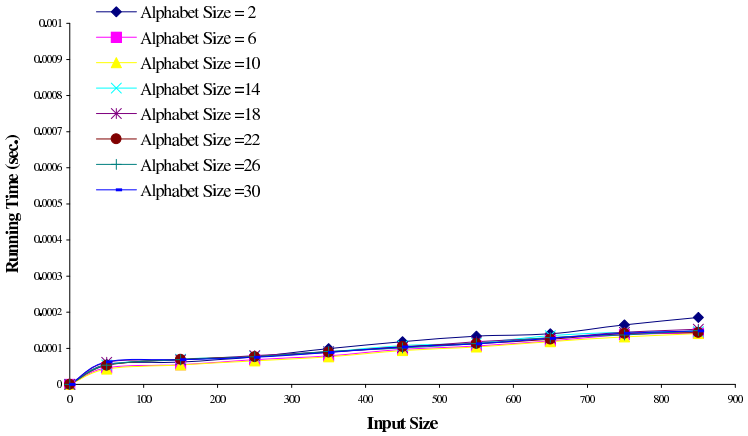


Fig. 5: Timing curves for the FAILURE-FUNCTION-2 procedure.

6. Computing the covers

In this section we present an algorithm (Fig. 6) 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 can use either one of the two failure function algorithms. Using the previously computed borders, the procedure that finds the covers of a given string x is as follows:

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

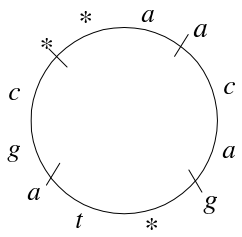
Fig. 6: FIND-COVERS algorithm.

THEOREM 1. *Given a string x that contains don't care symbols, we can find a longest cover u of x in linear expected time.*

PROOF. Finding the longest cover of a given string x can be achieved using FIND-COVERS algorithm. The overall expected running time for FIND-COVERS is $O(n)$. Stage 1 takes expected $O(n)$ time using either FAILURE-FUNCTION-1 or FAILURE-FUNCTION-2. Stage 2 also requires a linear expected running time. This is because the expected number of borders is bounded by a constant (Section 4) and checking each border can be done in $O(n)$ 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. 7). 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 Ben-Amram *et al.* 1994] does not work and we need a new approach.



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

Fig. 7: A circular string x and three substrings S1, S2, S3, as seen from a window of four characters length.

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.

8. 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.

Acknowledgements

C. S. Iliopoulos is partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants. Manal Mohamed is supported by an EPSRC studentship. W. F. Smyth is supported by a grant from the Natural Sciences & Engineering Research Council of Canada.

References

Abrahamson, K. 1987. Generalized String Matching. *SIAM Journal on Computing 16*, 6, 1039–1051.

Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffey D. 1974. *The Design & Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass, USA.

Apostolico, A. and Ehrenfeucht, A. 1993. Efficient Detection of Quasiperiodicities in Strings. *Theoretical Computer Science 119*, 2, 247–265.

Apostolico, A., Farach, M., and Iliopoulos, C. S. 1991. Optimal Superprimitivity Testing for Strings. *Information Processing Letters 39*, 1, 17–20.

Ben-Amram, A. M., Berkman, O., Iliopoulos, C. S., and Park, K. 1994. The Subtree Max Gap Problem with Application to Parallel String Covering. In *Proceedings of the 5th ACM-SIAM Symp. Discrete Algorithms*. Arlington, Virginia, 501–510.

Breslauer, D. 1992. An On-Line String Superprimitivity Test. *Information Processing Letters 44*, 6, 345–347.

Breslauer, D. 1994. Testing String Superprimitivity in Parallel. *Information Processing Letters 49*, 5, 235–241.

Fischer, M. and Paterson, M. 1974. String Matching and Other Products. In *Complexity of Computation SIAM-AMS Proceedings*, 113–125.

Iliopoulos, C. S., Moore, D. W. G., and Park, K. 1996. Covering a String. *Algorithmica 16*, 3, 288–297.

Iliopoulos, C. S. and Park, K. 1994. An Optimal $O(\log \log n)$ Time Algorithm for Parallel Superprimitivity Testing. *Journal of the Korean Information Science Society 21*, 8, 1400–1404.

Landau, G. M. and Viskin, U. 1989. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms 10*, 2, 157–169.

Li, Yin and Smyth, W. F. 2002. Computing the Cover Array in Linear time. *Algorithmica 32*, 1, 95–106.

Lothaire, M. 1983. *Combinatorics on Words*. Addison-Wesley, Reading, Mass, USA.

Moore, D. W. G. and Smyth, W. F. 1995. A correction to "Computing the Covers of a String in Linear Time". *Inform. Process. Lett. 54*, 2, 101–103.

Pevzner, P. A. 1990. *Statistical Analysis of Genetics Texts, Computer Analysis of Genetics Texts*. Nauka, Moscow.

Pevzner, P. A. 1993. *Overlapping Word Paradox and Conway Equation, Supercomputing, Bioinformatics, and Complex Genome Analysis, World Scientific*.

Pinter, R. 1985. Efficient String Matching with Don't-Care Patterns. In *Combinatorial Algorithms on Words*, NATO ASI. Springer-Verlag, 11–29.