APPROXIMATE PERIODICITY IN STRINGS

W. F. Smyth

School of Computing
Curtin University of Technology

Department of Computer Science & Systems
McMaster University

ABSTRACT

In many application areas (for instance, DNA sequence analysis), it becomes important to compute various kinds of "approximate period" of a given string y. Here we discuss three such approximate periods and the algorithms which compute them: an Abelian generator, a cover, and a seed. Let u be a substring of y. Then u is an *Abelian generator* of y iff y is a concatenation of substrings which are permutations of u; u is a *cover* of y iff every letter of y is contained in an occurrence of u in y; and u is a *seed* of y iff y is a substring of a string y' with cover u. Observe that, according to these definitions, y is an Abelian generator, a cover, and a seed of itself.

1 INTRODUCTION

Let A denote a nonempty (finite or infinite) set called the *alphabet*. An element of A is called a *letter*. Denote by A^+ the (infinite) set of all (finite) concatenations of the letters of A. We call the elements of A^+ *strings* and we say then that A^+ is the set of all possible nonempty strings over the alphabet A. For example, given $A = \{a, b\}$,

$$A^+ = \{a, b, a^2, ab, ba, b^2, \ldots\}$$

denotes the set of all strings on two letters a and b. Observe that in A^+ there are exactly m^n strings containing n letters, where $m = |A|$. Denoting by ϵ the empty string, we write $A^* = A^+ \cup \{\epsilon\}$, the set of all strings over A.

A string x that is a concatenation of $n \geq 0$ letters is said to have *length* $|x| = n$. In particular, a nonempty string x of length n is written

$$x = x[1]x[2]\cdots x[n],$$

abbreviated to $x[1..n]$. Given integers i and j satisfying $1 \leq i \leq j \leq n$, we say that $x[i..j] = x[i]x[i+1]\cdots x[j]$ is a *substring* of x. If $i > 1$ or $j < n$, $x[i..j]$ is called a *proper substring* of x. Clearly every substring of x is also a string; that is, an element of A^+. If x can be written as a concatenation uv of two (proper) substrings u and v, then $n = |u| + |v|$, u is said to be a *prefix* of x, and v is called a *suffix* of x. A string u that is both a prefix and a suffix of x is called a *border* of x.

If x has a border, it is called *periodic*; otherwise, x is said to be *primitive*. For example, $x = ababb$ is primitive. Let u denote a border of x of length j, where

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

$1 \le j \le n-1$; then the string $x[1..p]$, where $p = n - j$, is called the *j-period* of x and the integer $k = \lfloor n/p \rfloor$ is called the *j-exponent* of x. For example, observe that the string $x = abaab$ has only the single border $u = ab$ of length $j = 2$; thus x has 2-period $x[1..3] = aba$ and 2-exponent $\lfloor 5/3 \rfloor = 1$. On the other hand, $x = abaabaab$ has two borders ab and $abaab$ of lengths 2 and 5 respectively, giving rise to 2-period $x[1..6] = (aba)^2$, 2-exponent $\lfloor 8/6 \rfloor = 1$, 5-period $x[1..3] = aba$, and 5-exponent $\lfloor 8/3 \rfloor = 2$. Let u^* denote the border of x of maximum length j^*, where possibly $j^* = 0$ (in the case that x is primitive); then the j^*-period and the j^*-exponent of x are simply called the *period* and the *exponent*, respectively, of x. Thus a primitive string x has period x and exponent $k = 1$. A nonprimitive string is said to be *weakly periodic* iff it has exponent $k = 1$ and *strongly periodic* iff $k \ge 2$. Thus $abaab$ is weakly periodic and $abaabaab$ is strongly periodic. A strongly periodic string x is said to be a *repetition* (or *repetitive*) iff $n = pk$; in this case, we call the period of x its *generator*. For example, $ababab = (ab)^3$ has $n = 6$, $p = 2$, $k = 3$, and is therefore a repetition with generator ab.

The above definitions provide a basic vocabulary for a discussion of algorithms for finding patterns in strings. Section 2 classifies these patterns into three distinct categories (specific, generic, intrinsic); in particular, an algorithm for finding an important intrinsic pattern called the "normal form" is described. Section 3, the main part of this paper, examines various generic patterns and the algorithms for finding them; these patterns and their associated algorithms turn out to be related in interesting ways. Section 4 outlines open problems and future work.

2 A CLASSIFICATION OF PATTERNS

Most problems on strings reduce to finding some kind of pattern in them, but the nature of the pattern that is searched for varies widely. Traditionally, over the last 30 years, the pattern has generally been what we will call here *specific*; that is, the pattern is itself a given string, say u, and the search is for all occurrences of u in a given string x. For example, given $u = aaba$ and

$$x = abaabaabaab, \qquad \ldots(2.1)$$

the two overlapping occurrences of u would be encoded by the set $\{3, 6\}$, indicating that

$$u = x[3,6] = x[6,9].$$

The classical algorithms for finding specific patterns are due to Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77], both of which execute in time linear in the length $|x|$ of the string being searched, independent of the length of the pattern. Since 1977 literally dozens of variant, improved and hybrid algorithms have been proposed, some of the most recent of which are described in [CH92,GG92,WM92]. Further variation is derived from the fact that the pattern u can include "don't-care" letters each of which may be any element of A. Thus, for example, a search for $u = a*$ in (2.1), where $*$ means "either a or b", would yield matches

$$x[1..2] = x[4..5] = x[7..8] = x[10..11] = ab,$$
$$x[3..4] = x[6..7] = x[9..10] = aa.$$

Another common variation of the specific pattern matching problem is called *approximate pattern matching* (not to be confused with the "approximate periods" discussed in Section 3). Approximate pattern matching is particularly important in DNA sequence analysis; a best match is chosen based on a system of "scoring" which responds to the insertion of new letters into the pattern and the deletion/change of existing ones [GP90].

During the last 15 years interest has grown in finding patterns not specified as substrings, but defined rather in terms of their properties: we call such patterns *generic*. The classic example of a generic pattern is a repetition (defined in Section 1). The mathematical study of repetitions dates back to Thue [T06] at the turn of the century, but it is only since about 1980 that computer scientists have begun to consider algorithms to find all the repetitions in a given string x — what we shall call here the *R Problem*. Three such algorithms — using very different methods — have since been designed [C81,AP83,ML84], each executing in time $\Theta(n \log n)$, where $n = |x|$. These algorithms encode repetitions as triples (i, p, k), $k > 1$, signifying that
(a) $x[i..i+p-1]$ is *not* a repetition;
(b) $x[i..i+pk-1] = x[i..i+p-1]^k$;
(c) $x[i..i+p(k+1)-1] \neq x[i..i+p-1]^{k+1}$.
In example (2.1), the output triples are $\{(3,2,2), (6,2,2), (9,2,2)\}$ for a^2, $(1,6,3)$ for $(aba)^3$, $(2,6,3)$ for $(baa)^3$, and $(3,6,3)$ for $(aab)^3$. We shall have more to say about repetitions and other kinds of generic pattern in Section 3.

A third class of patterns consists of those whose existence is not in question: these patterns always exist in any string x, and so are called *intrinsic*. For an ordered alphabet, the classic example of an intrinsic pattern in x is the decomposition of x into "Lyndon words" [CFL58]. A string w is a *Lyndon word* if and only if, for every prefix u and suffix v such that $w = uv$, it is true that $w < vu$ in lexicographical order. Thus every single letter of A is a Lyndon word, and, for example, if $A = \{a, b\}$ with $a < b$, then ab and aab are Lyndon words, but not aa, ba or aba. It turns out [CFL58] that there is a unique decomposition of every string

$$x = w_1 w_2 \cdots w_m$$

into Lyndon words $w_1 \geq w_2 \geq \cdots \geq w_m$. For the example (2.1), the unique Lyndon decomposition is $x = w_1 w_2 w_3 w_4$, where

$$w_1 = ab;\ w_i = aab, \text{ for } i = 2, 3, 4.$$

The Lyndon decomposition of x can be computed in time $\Theta(n)$ [D83,IS94]; this decomposition is intimately connected with the idea of a "circular" string (see Section 3), and has unexpected applications to computer graphics [IS89] and graph theory [CB81].

What is perhaps the most important and widely-applicable intrinsic pattern is computable even on an unordered alphabet, as is now described. For any nonempty string x, let $k \geq 1$ be the greatest integer such that

$$x = (v'v)^k v' \qquad \ldots (2.2)$$

over all possible choices of the strings v' (possibly empty) and $v \neq \epsilon$. Then let v^* be the maximum length string v' which satisfies (2.2). Thus

$$x = (v^*v)^k v^*. \qquad \ldots(2.3)$$

We call the righthand side of (2.3) the *normal form* of x. In terms of the definitions of Section 1, it is easy to see that $(v^*v)^{k-1}v^*$ is the longest border of x, hence that $p = |v^*v|$; then $x[1..p]$ is the period and k the exponent. Therefore (2.3) may be rewritten

$$x = x[1..p]^k x[1..r], \qquad \ldots(2.4)$$

where $r = |v^*|$. Then x is primitive iff $k = 1$ and $r = 0$; weakly periodic iff $k = 1$ and $r > 0$; strongly periodic iff $k > 1$; and a repetition with generator $x[1..p]$ iff $k > 1$ and $r = 0$.

The normal form is an immediate consequence of the classical failure function computation. Let f denote a string of integers in which $f[i]$, $1 \leq i \leq n$, is the length of the longest border of $x[1..i]$. f is called the *failure function* of x and can be computed in time $\Theta(n)$ by a standard algorithm [AHU74]. Then

$$p = n - f[n]; \quad k = \lfloor n/p \rfloor; \quad r = n - pk. \qquad \ldots(2.5)$$

For the example (2.1), $n = 11$, $f[n] = 8$, $p = 3$, $k = 3$, and $r = 2$. We have just seen that the normal form allows us to identify a repetition and its generator; in the next section we shall see that it is also useful for the identification of other, more subtle generic patterns.

3 FINDING GENERIC PATTERNS

We begin by revisiting the R Problem: finding all the repetitions in a given string x of length $n > 0$. As we have seen, this problem can be solved in time $\Theta(n \log n)$; the question arises whether the time requirement is least possible. Consider first the 9 distinct squares in the string $x = a^6$:

$$x[1..2] = x[2..3] = x[3..4] = x[4..5] = x[5..6] = a^2;$$
$$x[1..4] = x[2..5] = x[3..6] = (aa)^2;$$
$$x[1..6] = (aaa)^2.$$

Generalizing this example slightly, it is easy to prove by induction that, for arbitrary n, there are $\lfloor n^2/4 \rfloor$ distinct squares in the string $x = a^n$, and so the (i, p, k) encoding of repetitions described in Section 2 is critical in reducing the size of the required output below $\Theta(n^2)$. In this example, all $\lfloor n^2/4 \rfloor$ repetitions would in fact be encoded by the single output $(1, 1, n)$, but it turns out that a Fibonacci string of length n actually requires $\Theta(n \log n)$ outputs in the (i, p, k) encoding [C81]. (A Fibonacci string is a string whose length n is a Fibonacci number, defined on $A = \{a, b\}$ as follows: $f_0 = b$, $f_1 = a$; $f_j = f_{j-1}f_{j-2}$ for every integer $j \geq 2$.) Indeed, as shown in [ML84], a much stronger statment holds, independent of any encoding of the output: even to recognize whether or not a given string contains a repetition requires time $\Omega(n \log n)$. Thus the three $\Theta(n \log n)$ algorithms are in fact asymptotically optimal, and we have

Theorem 3.1 Finding all the repetitions in x requires exactly $\Theta(n \log n)$ time. □

We now introduce a new kind of generic pattern. A nonempty string y is said to be an *Abelian repetition* iff for some integer $m > 1$,

$$y = u_1 u_2 \cdots u_m, \qquad \ldots (3.1)$$

where each u_i, $i = 1, 2, \ldots, m$, is a permutation of u_1. If in addition $m = m^*$ is the greatest integer satisfying (3.1), then each u_i, $1 \leq i \leq m^*$, is called an *Abelian generator* of y. For example, the string $y_1 = ababbaababba$ is an Abelian repetition corresponding to $m = 2, 3, 6$ with Abelian generators $\{ab, ba\}$ and $m^* = 6$; while $y_2 = abababab bbaa$ is an Abelian repetition corresponding to $m = 2, 3$ with Abelian generators $\{abab, bbaa\}$ and $m^* = 3$. Clearly every repetition is also an Abelian repetition, and so the *AR Problem* — that of finding all Abelian repetitions in a given string x — is a generalization of the R Problem. Observe however that not every generator is necessarily an Abelian generator: y_1 has generator $ababba$, not ab. At the same time, neither is it true that every Abelian generator is a generator: y_2 is not a repetition and has no generator.

The AR Problem is apparently more difficult than the R Problem [CS94]. The fastest known algorithm is an "obvious" one and requires $\Theta(n^2)$ time in all cases. Further, it has been shown that, for all known encodings of the output, there exist strings (specifically, Fibonacci strings again) which give rise to $\Theta(n^2)$ Abelian repetitions [CS94]. On the other hand, the best lower bound on the problem of recognizing whether or not x contains an Abelian repetition is $\Omega(n \log n)$, the same as the bound for the corresponding recognition problem on repetitions. Thus, the following conjecture remains open:

Conjecture 3.2 Finding all the Abelian repetitions in x requires $\Theta(n^2)$ time in the worst case. □

Even if this conjecture is true, it still remains an open problem whether or not there exists an $O(n^2)$ time AR algorithm which, when applied to some strings x, would require time somewhere between $\Omega(n \log n)$ and $O(n^2)$; such an algorithm would be of particular interest if its execution time were $\Theta(\max\{n \log n, r(x)\})$, where $r(x)$ denotes the size of the output.

The generator of an Abelian repetition is one kind of approximate period; we now consider another, quite different one called a "cover". Suppose u is a substring of a given string y. Then we say that u is a *cover* of y iff every element of y is contained in an occurrence of u in y. Thus $y_1 = abaababaab$ has covers y_1 and $abaab$, while $y_2 = abaababaaba$ has covers y_2, $(aba)^2$ and aba. A cover u of y is said to be *proper* iff $u \neq y$. Observe that a generator of a repetition y is a proper cover of y, so that the problem of finding all the proper covers, if any, of y (which we call the *PC Problem*) generalizes the problem of finding the generator, if any, of y. If y has a proper cover, we say that y is *coverable*. Then every repetition is coverable, and so the problem of finding all the coverable strings in a given string x (which we call the *CS Problem*) is a generalization of the R Problem.

It turns out [MS94,MS95] that the covers of a given string y can be characterized in terms of the normal form (2.4); that is, in terms of the parameters (2.5):

Theorem 3.3 Suppose a given string y has normal form $y[1..p]^k y[1..r]$. Then each cover of y is *either*
(a) $y[1..p]^j y[1..r]$, $j = 1, 2, \dots, k$; *or*
(b) (if $k > 1$) a proper cover of $y[1..p+r]$; *or*
(c) (if $k = 1$) a cover of $y[1..r]$ which also covers $y[1..p+r]$.

As an example of this theorem, consider $y = abababababaababababaabababa$ with normal form specified by $(p, k, r) = (7, 3, 5)$. The proper covers of y are $y[1..7]y[1..5]$, $y[1..7]^2y[1..5]$, together with (since $k > 1$) the proper covers of $y' = y[1..12] = ababaababab a$. y' in turn has normal form given by $(p', k', r') = (7, 1, 5)$ whose proper covers are (since $k' = 1$) those covers of $y'[1..r'] = ababa$ which also cover y'; that is, $ababa$ and aba. Observe that, since proper covers of y are necessarily also prefixes of y, the four proper covers found in this example can economically be output merely by specifying four prefix lengths; that is $\{3, 5, 12, 19\}$.

This example illustrates the way in which Theorem 3.3 leads directly to a classical divide-and-conquer algorithm for the PC Problem: finding all the covers of y is reduced essentially to finding all the covers of a specified proper prefix of y. Note that in Theorem 3.3(b) the prefix is of length less than $2|y|/3$, while in Theorem 3.3(c) the prefix is of length less than $|y|/2$. Thus the D&C algorithm requires at most $O(\log |y|)$ recursive steps to compute all the covers of y. The hard part of this algorithm arises from Theorem 3.3(c), where it is necessary to check that a cover of $y[1..r]$ also covers $y[1..p+r]$; however, it turns out that, with some $O(n)$ time preprocessing to compute a doubly-linked list of $O(n)$ integers (in addition to the n integers required for the failure function), this check, and in fact each step of the D&C algorithm, can be executed in constant time. We have then

Theorem 3.4 Let y denote a string of length n. Finding all the covers of y requires exactly $\Theta(n)$ time and $\Theta(n)$ additional space. $\square$

This method of handling the PC Problem suggests that it might be possible to compute all the covers of every prefix of a given string y (the *PPC problem*) in $\Theta(n)$ time, thus improving on the currently fastest algorithm [B92] that requires $\Theta(n \log n)$ time. Hence:

Conjecture 3.5 Finding all the covers of every prefix of y requires $\Theta(n)$ time.

We now briefly consider the CS Problem which, since it is a generalization of the R Problem, can at best be solved in time $\Omega(n \log n)$. In [AE90] an $O(n \log^2 n)$ time algorithm for the CS Problem was proposed, while in [MS94] it was suggested that the insights gained from Theorems 3.3 and 3.4 might give rise to a faster algorithm. Hence:

Conjecture 3.6 Finding all the coverable strings in x requires $\Theta(n \log n)$ time in the worst case. $\square$

We now turn to the final kind of approximate periodicity to be considered in this section. A substring u of y is said to be a *seed* of y iff y is a substring of a string y' with cover u. Thus, since y is a substring of itself, every cover of y is necessarily also a seed of y, and so the problem of finding all the proper seeds of y (which we call here the PS Problem) is a generalization of the PC Problem defined above. In [IMP93] a $\Theta(n \log n)$ time algorithm is presented for the solution of the PS Problem; however, recent results [IMPS94] encourage the conjecture that the PS problem can actually be solved in linear time. These results are outlined below.

We observe first that a given string y of length n may have $\Theta(n^2)$ seeds. For example, consider the string $y = (abc)^4ab$, whose normal form is specified by the parameters $(p, k, r) = (3, 4, 2)$. It is not hard to see that the seeds of this string are exactly all the substrings of y of length at least $p = 3$; we may enumerate these substrings by specifying first y itself, then $(abc)^4a$ and $(bca)^4b$, and then exactly $p = 3$ "rotations" of each prefix $y[1..j]$ of length $j = p, p+1, \ldots, n-p+1$. Thus, for $j = 3$, the corresponding seeds are the substrings $\{abc, bca, cab\}$, while for $j = n-p+1 = 12$, they are $\{(abc)^4, (bca)^4, (cab)^4\}$. Then the total number of distinct seeds in this example is

$$p(n-p+3/2) = p\big[(n-p+1)-p+1\big] + \big(1+2+\cdots+(p-1)\big).$$

By holding k and r constant while allowing p to increase indefinitely, an infinite family of such examples can be generated, for each of which there are exactly $p(n-p+3/2)$ seeds. Since $p \approx n/4$, it follows that the number of seeds of each member of this family is $\Theta(n^2)$.

To make these ideas both more general and more precise, we require some further definitions. First let a *rotation* of a nonempty string $y = y[1..n]$ be any substring of $yy[1..n-1]$ of length exactly n. We denote the rotations of y by the symbols $R_i(y)$, $i = 1, 2, \ldots, n$, where $R_i(y) = y[i..n]y[1..i-1]$; thus $R_1(y) = y$. A closely related idea is that of the *circular string* $C(y)$ of a given string y, formed by concatenating $y[1]$ at the right of $y[n]$. Alternatively, $C(y)$ may be thought of as the string of length n which contains each $R_i(y)$, $i = 1, 2, \ldots, n$, as a substring. In fact, observe that a string y' is a rotation of y if and only if $C(y) = C(y')$. For a circular string $C(y)$, a *cover* u is defined to be a substring of some $R_i(y)$ which is a cover (in the ordinary sense) of some substring of $yy[1..n-1]$ of length at least n. Observe in particular that every rotation of y is a cover of $C(y)$; a cover of $C(y)$ which is not a rotation of y is said to be *proper*. As an example, consider the string $y_1 = baaba$. The rotations of y_1 are

$$R_1 = baaba,\ R_2 = aabab,\ R_3 = ababa,\ R_4 = babaa,\ R_5 = abaab;$$

and, even though y_1 has no proper cover, $C(y_1)$ has the proper cover $u = aba$, since u is a cover of the substring $R_3 = ababa$ of $y_1y_1[1..4]$. Based on these definitions, the following result can now be proved [IMPS94]:

Theorem 3.7 Suppose a given string y has normal form $y[1..p]^k y[1..r]$. Then the seeds of y are exactly the following:
(a) all substrings of y of length at least p;

(b) all proper covers of $C(y[1..p])$. $\square$

The seeds specified by Theorem 3.7(a) are determined as a direct consequence of the $\Theta(n)$ time normal form computation; in fact, to describe these seeds, it suffices to output p, and so these seeds may be regarded as being computable in linear time. Thus the PS Problem is reduced to the problem of finding all the proper covers of a circular string (called here the *CC Problem*).

In order to discuss the CC Problem, we need two simple ideas. The first of these is the *normal form of a circular string* $C(y)$, which is defined to be the normal form of that rotation $R_j(y)$ whose normal form (in the ordinary sense) minimizes the value of p over all the rotations of y. Thus, conceptually, the normal form of a circular string can be determined by computing the normal form of all of its rotations. For the example y_1 given above, the normal form of $C(y_1)$ is just the normal form of R_3, determined by the parameters $(p, k, r) = (2, 2, 1)$. The second idea required is that of an "extension": given a string y, we say that vyu is an *extension* of y iff v is either a suffix of y or empty, and u is either a prefix of y or empty. We observe that, given a string y, there exists a set of extensions of every rotation of y which has the same set of covers that $C(y)$ has. This remark suggests the following outline of a linear time algorithm to solve the CC Problem [IMPS94]:

(1) Compute the normal form $z = R_j(y)$ of $C(y)$ (using modified failure function computations).
(2) Compute minimum extensions $E(z)$ of z which include a subset of the covers of $C(y)$.
(3) Use the Moore-Smyth algorithm [MS95] to compute the covers of $E(z)$.

We have then

Conjecture 3.8 Let y denote a string of length n. Finding all the covers of $C(y)$, hence finding all the seeds of y, requires exactly $\Theta(n)$ time and $\Theta(n)$ additional space. $\square$

4 CONCLUDING REMARKS

This paper has surveyed some of the most important kinds of pattern which are recognized and searched for in strings. In particular, generic patterns that correspond to some form of approximate periodicity have been discussed, as well as the algorithms that compute them. Interesting and suggestive relationships have been discovered among these patterns and their associated algorithms, and a number of open problems have been stated. Above all, it appears likely that there are many different kinds of approximate periodicity which have not yet been recognized, and which may well be efficiently computable. It is this possibility which motivates my own future work in this area.

REFERENCES

[**AHU74**] Alfred V. Aho, John E. Hopcroft & Jeffrey D. Ullman, *The Design & Analysis of Computer Algorithms*, Addison-Wesley (1974).

[**AE90**] A. Apostolico & Andrzej Ehrenfeucht, *Efficient Detection of Quasiperiodicities in Strings*, Tech. Report No. 90.5, The Leonardo Fibonacci Institute, Trento, Italy (1990).

[**AP83**] A. Apostolico & F. P. Preparata, **Optimal off-line detection of repetitions in a string**, *TCS 22* (1983) 297-315.

[**B92**] D. Breslauer, **An on-line string superprimitivity test**, *Inform. Process. Lett. 44* (1992) 345-347.

[**BM77**] R. S. Boyer & J. S. Moore, **A fast string searching algorithm**, *Commun. ACM 20* (1977) 213-216.

[**CFL58**] K. T. Chen, R. H. Fox & R. C. Lyndon, **Free differential calculus IV**, *Ann. of Math. 68* (1958) 81-95.

[**CB81**] C. J. Colbourn & K. S. Booth, **Linear time isomorphism problems for trees, interval graphs, and planar graphs**, *SIAM J. Comput. 10* (1981) 203-225.

[**CH92**] Richard Cole & Ramesh Hariharan, **Tighter bounds on the exact complexity of string matching**, *Proc. 33rd Annual IEEE Symposium on Foundations of Comp. Sci.* (1992) 600-609.

[**C81**] Max Crochemore, **An optimal algorithm for finding the repetitions in a word**, *IPL 12-5* (1981) 244-248.

[**CS94**] L. J. Cummings & W. F. Smyth, **Weak repetitions in strings**, *J. Comb. Math. & Comb. Computing*, to appear.

[**D83**] P. Duval, **Factoring words over an ordered alphabet**, *J. Algs. 4* (1983) 363-381.

[**GG92**] Zvi Galil & Raffaele Giancarlo, **On the exact complexity of string matching: upper bounds**, *SIAM J. Comput. 21-3* (1992) 407-437.

[**GP90**] Zvi Galil & Kunsoo Park, **An improved algorithm for approximate string matching**, *SIAM J. Comput. 19-6* (1990) 989-999.

[**IMP93**] C. S. Iliopoulos, Dennis Moore & Kunsoo Park, **Covering a string**, *Proc. Fourth Annual Symposium on Combinatorial Pattern Matching* (1993) 54-62.

[**IMPS94**] C. S. Iliopoulos, Dennis Moore, Kunsoo Park & W. F. Smyth, **Computing all the seeds of a string in linear time**, work in progress.

[**IS89**] C. S. Iliopoulos & W. F. Smyth, **PRAM algorithms for identifying polygon similarity**, *Springer-Verlag Lecture Notes in Computer Science 401*, H. Djidjev (ed.) (1989) 25-32.

[**IS94**] C. S. Iliopoulos & W. F. Smyth, **A fast average case algorithm for Lyndon decomposition**, *Internat. J. Computer Math. 57* (1995) 15-31.

[**KMP77**] D. E. Knuth, J. H. Morris & V. B. Pratt, **Fast pattern matching in strings**, *SIAM J. Comput. 6* (1977) 189-195.

[**ML84**] Michael G. Main & Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algs. 5* (1984) 422-431.

[**MS94**] Dennis Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *Inform. Process. Lett. 50* (1994) 239-246.

[**MS95**] Dennis Moore & W. F. Smyth, **Correction to: An optimal algorithm to compute all the covers of a string**, *Inform. Process. Lett. 54* (1995) 101-103.

[**T06**] A. Thue, **Über unendliche Zeichenreichen**, *Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7* (1906) 1-22.

[**WM92**] Sun Wu & Udi Manber, **Fast text searching allowing errors**, *Commun. ACM 35-10* (1992) 83-91.

ACKNOWLEDGEMENTS

This work was supported in part by Grant No. A8180 of the Natural Sciences & Engineering Research Council of Canada.