Computing Regularities in Strings: A Survey[⋆]

W. F. Smyth

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

Abstract. The aim of this survey is to provide insight into the sequential algorithms that have been proposed to compute exact "regularities" in strings; that is, covers (or quasiperiods), seeds, repetitions, runs (or maximal periodicities), and repeats. After outlining and evaluating the algorithms that have been proposed for their computation, I suggest possibly productive future directions of research.

1 Introduction

A central concern of Thue's 1906 paper [99], the founding document of combinatorics on words, was the occurrence (or, rather, non-occurrence) of periodicities in infinite strings on three letters: he showed that such strings can be constructed to contain no squares. In the intervening century, certainly thousands of research papers have been written by mathematicians and (over the last half-century) also computer scientists that relate in some way to periodicity, or its variants, in strings. A word that has recently been brought into service to describe these variants is "regularities" [52]. In this paper, to trim away most of those thousands of publications, we concentrate on the *computation* of regularities and, to sharpen the focus still more, with two caveats:

- We do not consider regularities that are "approximate" in any sense; for instance, those that allow errors (such as regarding $(abc)(abd)$ as an approximation of $(abc)^2$) or rearrangement ("Abelian" squares such as (abc)(cab) [31, 34]), or those defined on strings that contain "don't cares" ("wild cards", "holes") [13, 14] or that contain other subsets of the alphabet ("indeterminate" [98] or "degenerate" [56] strings). Thus our regularities are *exact*.
- We do not consider computations that are distributed or parallel; we confine ourselves to *sequential* algorithms.

The main reason for avoiding these distractions (though they are important, and deserve surveys of their own) is that the methods used to compute approximate regularites or to handle parallel computation are strikingly different; by sticking to sequential algorithms on exact regularities, at least a certain unity

⋆ This work was supported in part by the Natural Sciences & Engineering Research Council of Canada. The author thanks two referees for constructive suggestions that have materially improved this paper.

of methodology is achieved. This paper shares some common ground with the previous surveys [7, 96, 4] and updates some of their material; also of course it has many points of intersection with the "Bible" of combinatorics on words [73].

We consider two quite different kinds of computation:

- In Sections 3 & 4 we compute regularities that *characterize* the string as well as (usually) all its prefixes. These regularities are extensions of the idea of a "failure function" [3] or "border array" [97] that permits all the periods of every prefix of a given string to be compactly specified by a single array of integers. The methods used for these problems often make use of structures equivalent to suffix trees in order to achieve efficient execution.
- On the other hand, in Sections 5 & 6, we are computing regularities that occur *within* the string: essentially, repeating substrings (factors) that are constrained to be adjacent ("repetitions") or otherwise those that may be nonadjacent or overlapping ("repeats"). The algorithms proposed for these computations generally depend on preprocessing that computes the suffix tree of the string (up until a few years ago) or else the suffix array (currently).

Having hopefully given a bird's-eye view of these algorithms in Sections 3-6, I provide in Section 7 one man's perspective on directions that might be taken to lead to faster algorithms for regularities in the future.

2 Preliminaries

A ***string*** is a finite sequence of symbols (***letters***) drawn from some finite or infinite set Σ called the ***alphabet***. The alphabet size is $\sigma = |\Sigma|$. We write a string $\boldsymbol{x}$ in mathbold, and we represent it as an array $\boldsymbol{x}[1..n]$ for some $n \geq 0$ called the ***length*** of $\boldsymbol{x}$, also written $|\boldsymbol{x}|$. For $n = 0$, $\boldsymbol{x} = \boldsymbol{\varepsilon}$, the ***empty string***. If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ is said to be a ***prefix***, $\boldsymbol{v}$ a ***substring*** and $\boldsymbol{w}$ a ***suffix*** of $\boldsymbol{x}$; if $\boldsymbol{vw} \neq \boldsymbol{\varepsilon}$, $\boldsymbol{uw} \neq \boldsymbol{\varepsilon}$, $\boldsymbol{uv} \neq \boldsymbol{\varepsilon}$, respectively, then $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ is, respectively, a ***proper prefix***, ***proper substring***, ***proper suffix*** of $\boldsymbol{x}$. (What we have defined here as a "substring" is also often called a ***factor***.)

If $\boldsymbol{x} = \boldsymbol{x}[1..n]$ has a proper (though possibly empty) prefix $\boldsymbol{u}$ that is also a suffix of $\boldsymbol{x}$, then $\boldsymbol{u}$ is said to be a ***border*** of $\boldsymbol{x}$. If for some $p \in 1..n$, $\boldsymbol{x}[i] = \boldsymbol{x}[p{+}i]$ for every $i \in 1..n{-}p$, then $\boldsymbol{x}$ is said to have ***period*** p. Thus $\boldsymbol{x}$ always has the empty border $\boldsymbol{\varepsilon}$ and trivial period n. It is well-known, and easy to prove, that $\boldsymbol{x}$ has period p if and only if it has a border of length $n{-}p$. Of course one of the most useful tools in dealing with periodicity in strings is the "Periodicity Lemma" [35]. The ***border array*** $\beta = \beta_{\boldsymbol{x}}$ of a string $\boldsymbol{x}$ is an array of length n such that $\beta_{\boldsymbol{x}}[i]$ equals the length of the longest border of $\boldsymbol{x}[1..i]$ for every $i \in 1..n$. Since $\beta_{\boldsymbol{x}}[i] = b > 0$ implies that $\beta_{\boldsymbol{x}}[b]$ is the next largest border of $\boldsymbol{x}[1..i]$, it follows that $\beta_{\boldsymbol{x}}$ specifies *all* the borders, hence all the periods, of every prefix of $\boldsymbol{x}$. A simple $\Theta(n)$-time algorithm computes the border array of $\boldsymbol{x}$ [3, 97]. Here for example are the maximum borders $\beta_{\boldsymbol{x}}$ and corresponding minimum periods

$p_{\boldsymbol{x}}$ of the prefixes of a simple string:

$$\begin{array}{rl} & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\ \boldsymbol{x} = & a\ b\ a\ a\ b\ a\ b\ a \\ \beta_{\boldsymbol{x}} = & 0\ 0\ 1\ 1\ 2\ 3\ 2\ 3 \\ p_{\boldsymbol{x}} = & 1\ 2\ 2\ 3\ 3\ 3\ 5\ 5 \end{array} \tag{1}$$

A string $\boldsymbol{x}$ has ***quasiperiod*** $q < n$ (and is accordingly called a ***quasiperiodicity***) if and only if there exists a string $\boldsymbol{u} = \boldsymbol{u}[1..q]$, called a ***cover*** of $\boldsymbol{x}$, such that every position of $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$. Thus a cover must also be a border of $\boldsymbol{x}$. If for some integer $k > 1$ there exists a set $\mathcal{C} = \{\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots, \boldsymbol{u_t}\}$ of strings, each of length k, such that every position in $\boldsymbol{x}$ lies within an occurrence of some element of $\mathcal{C}$, then $\mathcal{C}$ is said to be a ***k-cover*** of $\boldsymbol{x}$, a ***minimum k-cover*** if for fixed k, t is least possible. The string (1) has quasiperiod 3 and cover aba, and therefore for $k = 3$ has a minumum 3-cover of cardinality $t = 1$; however $\boldsymbol{x}$ also has a minimum 2-cover $\mathcal{C} = \{ab, ba\}$ of cardinality $t = 2$. In Section 3 we discuss algorithms related to covers and k-covers.

In order to define seeds of a given string $\boldsymbol{x}$, we consider an ***extension*** $\boldsymbol{w} = \boldsymbol{x_L x x_R}$ of $\boldsymbol{x}$, where $\boldsymbol{x_L x}$ is a ***left extension***, $\boldsymbol{x x_R}$ a ***right extension***. Then given a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$, we say that

- $\boldsymbol{u}$ is a ***left seed*** of $\boldsymbol{x}$ if it is a cover of some right extension of $\boldsymbol{x}$;
- $\boldsymbol{u}$ is a ***right seed*** of $\boldsymbol{x}$ if it is a cover of some left extension of $\boldsymbol{x}$;
- $\boldsymbol{u}$ is a ***seed*** of $\boldsymbol{x}$ if it is a cover of some extension of $\boldsymbol{x}$.

Thus any cover of $\boldsymbol{x}$ is trivially a seed with $\boldsymbol{x_L} = \boldsymbol{x_R} = \boldsymbol{\varepsilon}$; similarly, any left or right seed of $\boldsymbol{x}$ is trivially a seed. Observe that if $\boldsymbol{u}$ is a seed of $\boldsymbol{x}$, then we may assume WLOG that the length of any corresponding extension is strictly less than $|\boldsymbol{u}|$. In (1) $abaab$ and $ababa$ are left and right seeds, respectively, of $\boldsymbol{x}$, with $\boldsymbol{x_R} = ab$, $\boldsymbol{x_L} = ab$, respectively. Notice that a seed may provide information about the periodicity of a string that is not available from a cover; for example, $\boldsymbol{x} = abcabcabca$ is quasiperiodic with quasiperiod 4 and cover $abca$, but this does not describe the period 3 that is implied by any of the seeds abc, bca, cab. We discuss algorithms related to seeds in Section 4.

A ***repeating substring*** in $\boldsymbol{x}$ is a proper nonempty substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that occurs more than once — for example, $\boldsymbol{u} = aba$ in (1). A ***repeat*** in $\boldsymbol{x}$ is a tuple

$$M_{\boldsymbol{x},\boldsymbol{u},r} = \{\boldsymbol{u}; i_1, i_2, \ldots, i_r\}, \tag{2}$$

where $\boldsymbol{u}$ is the repeating substring that occurs at positions $i_1, i_2, \ldots, i_r$, with $1 \le i_1 < i_2 < \ldots < i_r \le n$. A repeat is said to be ***complete*** if $\boldsymbol{u}$ occurs *exactly* r times in $\boldsymbol{x}$. For example,

$$M_{\boldsymbol{x},ab,3} = \{ab; 1, 4, 6\} \text{ and } M_{\boldsymbol{x},aba,3} = \{aba; 1, 4, 6\}$$

are both complete repeats in (1). A repeat $M_{\boldsymbol{x},\boldsymbol{u},r}$ is said to be ***nonextendible*** (NE for short) if there exists no repeat $M_{\boldsymbol{x},\boldsymbol{v},r}$ such that $\boldsymbol{u}$ is a proper substring of $\boldsymbol{v}$; otherwise, ***extendible***. Normally, we are interested in complete NE repeats.

Perhaps of particular interest are ***supernonextendible*** (SNE) repeats: NE repeats $M_{\boldsymbol{x},\boldsymbol{u},r}$ such that $\boldsymbol{u}$ is not a substring of any other repeating substring in $\boldsymbol{x}$. In (1) both $M_{\boldsymbol{x},aba,3}$ and $M_{\boldsymbol{x},ab,3}$ are complete, but $M_{\boldsymbol{x},aba,3}$ is SNE, while $M_{\boldsymbol{x},ab,3}$ is extendible (not NE). Note that for the purpose of computer output a repeat (2) is fully specified by an $(r{+}1)$-tuple $(p, i_1, i_2, \ldots, i_r)$ of integers, where p is the length of the repeating substring $\boldsymbol{u}$. Algorithms that compute repeats are discussed in Section 6.

A ***repetition*** in $\boldsymbol{x}$ is a repeat (2) in which the occurrences of the repeating substring $\boldsymbol{u}$ are constrained to be adjacent (thus $i_{j+1} - i_j = |\boldsymbol{u}|$ for every $j \in 1..r{-}1$) and ***maximal*** (thus $\boldsymbol{x}[i{-}|\boldsymbol{u}|..i{-}1] \neq \boldsymbol{u}$ and $\boldsymbol{x}[i{+}r|\boldsymbol{u}|..i{+}(r{+}1)|\boldsymbol{u}|{-}1] \neq \boldsymbol{u}$). Of course $\boldsymbol{x}$ itself may be a repetition; if a string or substring is *not* a repetition, we say that it is ***primitive***. A repetition is fully specified by a triple (i, p, r), where $\boldsymbol{u} = \boldsymbol{x}[i..i{+}p{-}1]$ is the repeating substring, $p = |\boldsymbol{u}|$ the ***period*** of the repetition $\boldsymbol{u}^r$, and r the number of occurrences of $\boldsymbol{u}$ (or ***exponent*** of the repetition). If $r = 2$, the repetition is a ***square***. We assume throughout this paper that the repeating substring $\boldsymbol{u}$ is itself primitive — in other words, that $|\boldsymbol{u}|$ is the least possible period of the repetition. Thus to describe the repetition $\boldsymbol{x} = aaaa$, we write $(i, p, r) = (1, 1, 4)$ rather than $(1, 2, 2)$.

A ***run*** [97] (or ***maximal periodicity*** [74]) in $\boldsymbol{x}$ is a 4-tuple (i, p, r, t), where (i, p, r) is a repetition, $(i-1, p, r)$ is *not* a repetition, and $t \in 0..p-1$ is the maximum integer such that $\boldsymbol{x}[i{+}rp..i{+}rp{+}t{-}1] = \boldsymbol{u}[1..t]$. We call t the ***tail*** of the run. In (1) both $(3, 1, 2)$ and $(1, 3, 2)$ are repetitions (also runs with $t = 0$), while $(4, 2, 2, 1)$ is a run that implies two repetitions $(4, 2, 2)$ and $(5, 2, 2)$. (Note that $(5, 2, 2, 0)$ is *not* a run because $(4, 2, 2)$ is a repetition.) In general, computing all the runs determines all the repetitions. In Section 5 we discuss the computation of repetitions and runs.

We conclude this section with a brief mention of data structures that are computed in the preprocessing phase of many of the algorithms described below.

The ***suffix tree*** $\mathrm{ST}_{\boldsymbol{x}}$ of a string $\boldsymbol{x}[1..n]$ on an alphabet of size σ is a compacted trie [43] built on the suffixes of $\boldsymbol{x}$ (Figure 1 shows the suffix tree for the example string (1) with the starting positions of the suffixes occurring as leaf nodes in increasing lexicographic order). Several algorithms exist [101, 80, 100] to compute $\mathrm{ST}_{\boldsymbol{x}}$ in time $O(n \log \sigma)$ (thus $O(n \log n)$ for $\sigma \in O(n)$), while an impractical but influential one [33] (see also [97, pp. 126–136]), the model for practical recursive linear-time suffix array algorithms, computes $\mathrm{ST}_{\boldsymbol{x}}$ in $O(n)$ time independent of alphabet size — provided however that the letters of the alphabet can be treated as integers and so be sorted in linear time. The suffix tree has “myriad virtues” [6], including pattern-matching in time proportional to pattern length, and easy access to repetitions, repeats, and the longest common prefix of substrings; however, because of the need to use pointers and to store a search structure at each node, the space requirement, though linear in n, is nevertheless large, especially for large alphabets. In some cases space requirements can be reduced [70, 42, 38].

The ***suffix array*** $\mathrm{SA}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ is defined by $\mathrm{SA}_{\boldsymbol{x}}[i] = j$, $1 \leq i \leq n$, where $\boldsymbol{x}[j..n]$ is the i^{th} smallest suffix of $\boldsymbol{x}$ in lexicographic order. Since its introduction in 1990

$$\begin{array}{rcccccccccc}
 & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\boldsymbol{x} & = & a & b & a & a & b & a & b & a \\
\mathrm{SA}_{\boldsymbol{x}} & = & 8 & 3 & 6 & 1 & 4 & 7 & 2 & 5 \\
\mathrm{LCP}_{\boldsymbol{x}} & = & 0 & 1 & 1 & 3 & 3 & 0 & 2 & 2 \\
\mathrm{LPF}_{\boldsymbol{x}} & = & 0 & 0 & 1 & 3 & 2 & 3 & 2 & 1 \\
\mathrm{QSA}_{\boldsymbol{x}} & = & 0 & 0 & 1 & 1 & 2 & 4 & 5 & 6 \\
\mathrm{BWT}_{\boldsymbol{x}} & = & b & b & b & \$ & a & a & a & a
\end{array}$$

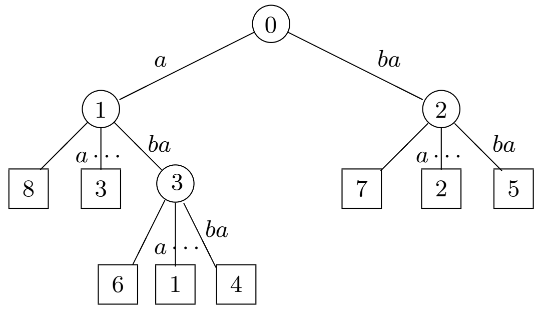

Fig. 1. Suffix tree, suffix array, LCP/LPF/QSA/BWT arrays

[76, 77], and especially over the last 10 years, the suffix array has replaced the suffix tree as the data structure of choice for string algorithms. Requiring just $4n$ bytes of storage in uncompressed form, it gradually became clear [64, 60, 62] that $\mathrm{SA}_{\boldsymbol{x}}$ could be computed both in linear time and more quickly than $\mathrm{ST}_{\boldsymbol{x}}$, also that it could be used in most cases at least as efficiently in algorithms [1]. The survey [91] gives an overview of suffix array construction algorithms (SACAs) up to 2007; the current algorithm of choice [85] requires only $5n$ bytes of space (for $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$), executes in linear time, and is fastest in practice [83]. Abouelehoda *et al.* [1] describe an "enhanced" suffix array $\mathrm{ESA}_{\boldsymbol{x}}$.

Often used in conjunction with $\mathrm{SA}_{\boldsymbol{x}}$ is $\mathrm{LCP}_{\boldsymbol{x}}$, the ***longest common prefix*** array, defined by

$$\mathrm{LCP}_{\boldsymbol{x}}[i] = lcp\{\mathrm{SA}_{\boldsymbol{x}}[i-1], \mathrm{SA}_{\boldsymbol{x}}[i]\},$$

for $i \in 2..n$. See Figure 1. Like the suffix array, $\mathrm{LCP}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time [61, 78, 92, 59], but with varying working storage requirements. Perhaps [92] at about $6n$ bytes provides the best trade-off between speed and storage.

A newly discovered data structure has turned out to be useful in various contexts. The ***longest previous factor*** (LPF) array was introduced by Crochemore & Ilie [25] (also under a different name by Franek *et al.* [37]): for any position i in $\boldsymbol{x}$, $\mathrm{LPF}_{\boldsymbol{x}}[i]$ is the length of the longest factor of $\boldsymbol{x}$ starting at i that occurs previously in $\boldsymbol{x}$. It turns out that LPF is a permutation of LCP, also that the LZ factorization (see below) can be easily computed from LPF [25]. Associated with LPF is the ***quasi suffix array*** QSA [37]: for every position $i \in 1..n$, $\mathrm{QSA}_{\boldsymbol{x}}[i] = 0$ if $\mathrm{LPF}_{\boldsymbol{x}}[i] = 0$, while for $\mathrm{LPF}_{\boldsymbol{x}}[i] > 0$, $\mathrm{QSA}_{\boldsymbol{x}}[i] = j$ for some $j \in 1..i{-}1$ such that

$$\boldsymbol{x}[j..j{+}\mathrm{LPF}[i]{-}1] = \boldsymbol{x}[i..i{+}\mathrm{LPF}[i]{-}1].$$

See Figure 1 for examples. It turns out that LPF and QSA together provide exactly the information required to compute the LZ factorization.

A data structure useful for the computation of repeats is the ***Burrows-Wheeler Transform*** or BWT [18]: for $\mathrm{SA}_{\boldsymbol{x}}[i] > 1$, $\mathrm{BWT}_{\boldsymbol{x}}[i] = \boldsymbol{x}\big[\mathrm{SA}_{\boldsymbol{x}}[i]{-}1\big]$, while for i such that $\mathrm{SA}_{\boldsymbol{x}}[i] = 1$, $\mathrm{BWT}_{\boldsymbol{x}}[i] = \$$, a special sentinel letter. See Figure 1. Due to its many applications — for instance, to data compression, index structures, and pattern-matching — the BWT has been intensively studied in recent years [2].

Finally we describe a data structure originally proposed for data compression, but that has turned out to have many other uses, especially for computing repetitions, but recently also for computing seeds. A factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ is ***LZ*** (for Lempel-Ziv [71, 102]) if and only if each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2 \cdots w_{j-1}}$; or otherwise
(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2 \cdots w_j}$.

We observe that $\boldsymbol{w_1} = \boldsymbol{x}[1]$, further that a factor $\boldsymbol{w_j}$ may overlap with its previous occurrence in $\boldsymbol{x}$: for the string $\boldsymbol{x} = abaababaab$, the LZ factorization is given by $\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = abaab$. The recent survey by Al-Hafeedh *et al.* [4] provides a comprehensive analysis and evaluation of LZ factorization algorithms in the context of the computation of repetitions.

3 Covers & *k*-Covers

In 1990 Apostolico and Ehrenfeucht introduced the idea of quasiperiod [8], later extended in [9]. Given a quasiperiodicity $\boldsymbol{w} = \boldsymbol{x}[i..j]$ of quasiperiod q in $\boldsymbol{x} = \boldsymbol{x}[1..n]$ (hence covered by $\boldsymbol{u} = \boldsymbol{x}[i..i{+}q{-}1] = \boldsymbol{x}[j{-}q{+}1..j]$), they defined $\boldsymbol{w}$ to be ***maximal*** if

- there exists no other quasiperiodicity $\boldsymbol{w'} = \boldsymbol{x}[i'..j']$ of the same quasiperiod q such that $\boldsymbol{w}$ is a proper substring of $\boldsymbol{w'}$; and
- for $j < n$, where $\lambda = \boldsymbol{x}[j{+}1]$, $\boldsymbol{u'} = \boldsymbol{u}\lambda$ does not cover $\boldsymbol{w}\lambda$.

They described an $O(n \log^2 n)$-time algorithm to compute all the maximal quasiperiodicities in $\boldsymbol{x}$. Later two other algorithms were published (Iliopoulos & Mouchard [55], Brodal & Pedersen [17]) that solved the problem in $O(n \log n)$ time, eventually shown by Groult & Richomme [45] to be optimal based on the construction of an infinite set S of strings such that every $\boldsymbol{z} \in S$ contains $O(|\boldsymbol{z}| \log |\boldsymbol{z}|)$ maximal quasiperiodicities. All the algorithms proposed to date for this problem make use of suffix trees, and so generally require a large amount of computer memory. This problem is the natural extension of the runs (maximal periodicities) problem described in Section 5.

A string is said to be ***primitive*** if it is not a repetition, ***superprimitive*** if it has no quasiperiod (not coverable by a single cover). In 1991 Apostolico *et al.* [10] described a recursive $O(n)$-time algorithm to compute the minimum cover of a string, if it has a cover; otherwise, to return "superprimitive". A year later Breslauer [15] published an on-line linear-time algorithm to compute the minimum cover of *each prefix* of a string. Then Moore & Smyth [81, 82] described a linear-time algorithm to compute *all* the covers of a string. Finally Li & Smyth [72] published an on-line linear-time algorithm that computes the ***cover array*** — an array $\gamma_{\boldsymbol{x}}$, analogous to the border array, that specifies all the covers of every prefix of a string $\boldsymbol{x}$ (zero if no cover exists). For example:

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} = & a & b & a & b & a & a & b & a & b & a \\ \gamma_{\boldsymbol{x}} = & 0 & 0 & 0 & 2 & 3 & 0 & 0 & 3 & 0 & 5 \end{array}$$

This array tells us that $\boldsymbol{x}$ has cover $\boldsymbol{u} = ababa$ of length 5, but also, since $\gamma[\gamma[10]] = \gamma[5] = 3$, cover $\boldsymbol{v} = aba$ of length 3.

The minimum k-cover problem was introduced in [57], where (incorrect) algorithms were given for its exact solution in polynomial time. In [23] Cole *et al.* showed by reduction to 3-SAT that the corresponding decision problem (whether there exists a k-cover of $\boldsymbol{x}$ of given cardinality) is NP-complete for every $k \geq 2$. It was shown further that the decision problem is a special case of the set cover problem, hence that the minimum k-cover can be computed to within a logarithmic factor by an efficient greedy algorithm. Two such $O(n \log n)$-time greedy algorithms were described, each making use of Crochemore's repetitions algorithm [24] as a preprocessor. More recently, Iliopoulos *et al.* [53] showed by reduction to the k-bounded set cover problem that a still closer approximation to the k-cover could be achieved in polynomial time.

In [47] Guo *et al.* investigate a problem with a superficial similarity to k-covers that however turns out to be easier: given $\boldsymbol{x}$ and an integer λ, find all sets S of substrings of $\boldsymbol{x}$, each of the same length, say k, that cover $\boldsymbol{x}$, subject to the constraint that $|S| = \lambda$. Thus it is required to find all sets of k-covers of $\boldsymbol{x}$ of cardinality exactly λ: the ***λ-covers*** problem. Their algorithm considers pairs (λ, k) for increasing values of k and executes in time $O(n^2)$. It makes use of ***successive refinement*** ("partitioning"), by which substrings of length k are extended ("refined") into substrings of length $k+1$; using the technique discovered by Hopcroft [49], and as we shall see used also in Crochemore's repetitions algorithm [24], the refinement of all substrings can be accomplished in $O(n \log n)$ time. The application of this technique is described in detail in [97, pp. 331–340], where also it is explained that a refinement is essentially a form of suffix tree. A significant drawback of the algorithm proposed in [47] is that it requires the alphabet size σ to be regarded as a constant; furthermore the constant of proportionality hidden inside the $O(n^2)$ is at least σ^{λ}.

Also in [47] Guo *et al.* consider a ***generalized λ-covers*** problem, where the requirement that the λ covering substrings be all of the same length is dropped. It turns out, surprisingly, that the time complexity is unaffected: the generalized problem can also be solved in time $O(n^2)$, though still of course with the same drawbacks.

4 Seeds

The seeds problem was introduced in 1996 by Iliopoulos *et al.* [54]. They described an $O(n \log n)$-time algorithm to compute all the seeds of a given string $\boldsymbol{x}[1..n]$ (that was however completed and corrected 15 years later in [22]). The method was again based on successive refinement, and for a long time it was not clear that a more time-efficient solution could be found, since the number of seeds can exceed n. For example, the 10 seeds of the string (1) of length $n = 8$ are

$$aba, abaab, baaba, aabab, ababa, (aba)^2, baabab, aababa, abaabab, baababa.$$

In fact it took 16 years to improve on the original all-seeds algorithm. In [65] Kociumaka *et al.* propose a complex algorithm that makes use of the LZ factorization to compute all the seeds of $\boldsymbol{x}$ in $\Theta(n)$ time.

In [46] Guo *et al.* consider the λ***-seeds*** problem, a straightforward generalization of the λ-covers problem. Using much the same methodology, they propose an algorithm whose time complexity is $O(n^2)$, again rather surprisingly the same as for the original λ-covers. Again σ must be constant and the constant of proportionality is at least σ^λ.

Very recently there has been considerable interest in analogues of the cover array modified for (left or right) seeds; that is, arrays $\mathrm{S}[1..n]$ such that $\mathrm{S}[i]$ gives the length of the (left or right) seed of $\boldsymbol{x}[1..i]$. More precisely, four variants have been considered:

- $\mathrm{RS}_{\mathrm{max}}[i]$ is the longest right seed of $\boldsymbol{x}[1..i]$;
- $\mathrm{RS}_{\mathrm{min}}[i]$ is the shortest right seed of $\boldsymbol{x}[1..i]$;
- $\mathrm{LS}_{\mathrm{max}}[i]$ is the longest left seed of $\boldsymbol{x}[1..i]$;
- $\mathrm{LS}_{\mathrm{min}}[i]$ is the shortest left seed of $\boldsymbol{x}[1..i]$.

In [21] algorithms are described to compute the $\mathrm{RS}_{\mathrm{max}}$ and $\mathrm{RS}_{\mathrm{min}}$ arrays in time $O(n)$ and $O(n \log n)$, respectively; in [22] linear-time algorithms to compute both $\mathrm{LS}_{\mathrm{max}}$ and $\mathrm{LS}_{\mathrm{min}}$ are proposed. The latter paper also describes a linear-time algorithm that uses $\mathrm{SA}_{\boldsymbol{x}}$ to check whether $\boldsymbol{x}$ has a seed of length k; this algorithm is then applied to compute in $O(n^2)$ time the array $\mathrm{S}_{\mathrm{min}}[1..n]$ giving the shortest *seed* of every prefix of $\boldsymbol{x}$.

5 Repetitions & Runs

Along with various algorithms for pattern-matching, the computation of repetitions was an early focus of computer scientists. Thus there are three "classical" repetitions algorithms, each optimal, each executing in $O(n \log n)$ time, but very different in approach:

- **Crochemore** [24] (see also [97, pp. 331–340]). As mentioned earlier, this algorithm makes use of a refinement technique that is essentially a suffix tree implementation. Crochemore showed that the Fibonacci string $\boldsymbol{f_r}$ ($\boldsymbol{f_0} = b$,

$\boldsymbol{f_1} = a$, $\boldsymbol{f_r} = \boldsymbol{f_{r-1}}\boldsymbol{f_{r-2}}$ for $r > 1$) contains $O(|\boldsymbol{f_r}|\log|\boldsymbol{f_r}|)$ repetitions, thus establishing the optimality of his algorithm (see also [97, pp. 76–85]). The data structures required for implementation are complex and space-consuming; however, careful implementation [38] not only reduces additional space to $13n$ words (integers), but permits output of all the runs and all the distinct squares in $\boldsymbol{x}$ as a byproduct. The implementation [39] executes in linear time on strings that occur in practice (for example, DNA, protein sequences, English text, source/executable code, Internet webpages) and may indeed be faster than any other runs algorithm proposed to date.

- **Apostolico & Preparata** [11]. This algorithm computes $\mathrm{ST}_{\boldsymbol{x}}$, which is used to construct a data structure called the leaf tree. The alphabet is constrained to be finite.
- **Main & Lorentz** [75] (see also [97, pp. 340–347]). In some respects the most interesting of these algorithms, the Main & Lorentz approach uses a divide-and-conquer technique that recursively splits each string into halves. At each step three kinds of repetitions are identified: those beginning and ending in the first half, those beginning and ending in the second half, and those that overlap. The output consists of repetitions and some of their cyclic shifts, thus foreshadowing the idea of a run or maximal periodicity. Unlike the other two repetitions algorithms, it is not required that the alphabet be ordered.

One might imagine that with three optimal algorithms for repetitions, there would be little more to say. But in 1989 Main [74] published an algorithm that computed "leftmost" runs in linear time. Ten years later Kolpakov & Kucherov [66, 67] (see also [97, pp. 350–358]) showed how to compute the remainder of the runs in time proportional to their number, and furthermore proved that the total number of runs in $\boldsymbol{x}[1..n]$ was at most $k_1 n - k_2\sqrt{n}\log n$ for some constants k_1, k_2. In other words, all repetitions could *implicitly* be reported in linear time.

The trouble was, the proof of the linearity of the runs was not constructive, and so the magnitudes of k_1 and k_2 are not specified in any way, even though [67] provided convincing experimental evidence (and conjectured) that the maximum number of runs (usually denoted $\rho(n)$) was at most n. The upper bound on $\rho(n)/n$ has since been successively shown to be 5.0 [93], 3.48 [88], 1.60 [26], 1.49 [44], and finally 1.029 [29], the last achieved with the aid of three years of CPU time on a network of high-performance computers. Meanwhile, the lower bound has gone from 0.92705 [40] to 0.9445756 [79], then to 0.944575712 [95]. More important from an algorithmic point of view, Puglisi & Simpson showed [87] that the *expected* value of $\rho(n)/n$ is about 0.4 for a binary alphabet, less than 0.05 for English text. Runs in strings are normally sparse.

Despite this sparsity, available methods for computing runs use heavy preprocessing that makes no use of combinatorial insights that might lead to algorithmic short cuts. As [4] explains in considerable detail, all competitive algorithms that compute the runs in a string $\boldsymbol{x}$ first compute $\mathrm{LZ}_{\boldsymbol{x}}$ by constructing some form of the suffix array — $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{ESA}_{\boldsymbol{x}}$ or $\mathrm{QSA}_{\boldsymbol{x}}$ — followed usually by $\mathrm{LCP}_{\boldsymbol{x}}$ or $\mathrm{LPF}_{\boldsymbol{x}}$. Over all the LZ algorithms considered [1, 19, 20, 25, 28, 27, 86], the preprocessing consumes at least 80% of the time required to compute $\text{LZ}_{\boldsymbol{x}}$. Once the LZ factorization has been computed, the runs are then computed by applying the algorithms of Main [74] and Kolpakov & Kucherov [67], as noted above; the time required for these procedures is also small with respect to preprocessing time. For more precise descriptions of LZ algorithms we refer the reader to [4].

In Section 7 we discuss approaches based on combinatorial analysis that might lead to greatly improved algorithms for computing runs.

6 Repeats

We identify four problems, the first two general in nature, the final two motivated particularly by applications in bioinformatics:

P1 Compute all NE (nonextendible) repeats in a given string $\boldsymbol{x}$.
P2 Compute all SNE (supernonextendible) repeats in $\boldsymbol{x}$.
P3 Compute all NE repeats in $\boldsymbol{x}$ that satisfy some constraint on the "gap" between repeating substrings.
P4 Compute all NE repeats in a set of strings ("multirepeats") that may also be required to satisfy a gap constraint.

In terms of this classification, the contributions to date include the following:

P1.1 In 1997 Gusfield [48, pp. 143 ff.] described an algorithm that, using $\text{ST}_{\boldsymbol{x}}$, computes all *pairs* of NE repeats in $\boldsymbol{x}$ in time $O(\sigma n+q)$, where q is the number of pairs output.
P1.2 In 2004 Abouelhoda *et al.* [1] solve the same problem with the same time complexity but using $\text{SA}_{\boldsymbol{x}}$ rather than $\text{ST}_{\boldsymbol{x}}$, thus less space. In 2003 Franek *et al.* [41] compute complete NE repeats in $\Theta(n)$ time, but their algorithm requires computation of suffix arrays both for $\boldsymbol{x}$ and the reversed $\boldsymbol{x}$, and though the output is $O(n)$, the repeats are not specified in their natural left-to-right order in $\boldsymbol{x}$.
P1.3 In 2007 Narisawa *et al.* [84] published a $\Theta(n)$-time algorithm that uses $\text{SA}_{\boldsymbol{x}}$ to compute all "substring equivalence classes", including the complete NE repeats, in $\boldsymbol{x}$.
P1.4 In 2008 Puglisi *et al.* [89, 90] published four variants of an algorithm that uses $\text{SA}_{\boldsymbol{x}}$, $\text{LCP}_{\boldsymbol{x}}$ and $\text{BWT}_{\boldsymbol{x}}$ to compute complete NE repeats $M_{\boldsymbol{x},\boldsymbol{u},r}$ such that the repeating substring $\boldsymbol{u}$ has a length that is at least some user-prescribed minimum. According to experiments described in [90], all variants are faster than previous algorithms; two of them are guaranteed to execute in linear time independent of alphabet size. Their output consists of ranges of positions in the suffix array; they propose postprocessing to reexpress the output (particularly on the DNA alphabet of four letters) into pairs or other convenient arrangements.
P1.5 Very recently Ilie & Smyth [50] have provided a different perspective on repeats. They establish a duality between minimum-length unique substrings (that is, those whose every substring is repeating) and maximum-length repeating substrings (that is, those whose every superstring is unique) in a

string $\boldsymbol{x}$. They show how minimum unique substrings and maximum repeating substrings cover any string, and they describe very simple, linear-time algorithms that use $\mathrm{SA}_{\boldsymbol{x}}$ and $\mathrm{LCP}_{\boldsymbol{x}}$ to compute one or the other.

P2.1 [48] and [1] also describe efficient algorithms to compute SNE repeats using $\mathrm{ST}_{\boldsymbol{x}}$ and $\mathrm{SA}_{\boldsymbol{x}}$, respectively; the former requires time $O(n \log \sigma)$, the latter $O(n+\sigma)$. For $\sigma \in O(n)$, these times become $O(n \log n)$ and $O(n^2)$, respectively.

P2.2 In [90] two SNE repeats algorithms are described, both very fast, both based on precomputation of $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{LCP}_{\boldsymbol{x}}$ and $\mathrm{BWT}_{\boldsymbol{x}}$; one of them, slightly slower in practice, guarantees $\Theta(n)$ processing time independent of alphabet size.

P3.1 In 2000 Brodal *et al.* [16] described an algorithm that used a modified $\mathrm{SA}_{\boldsymbol{x}}$ ("binary suffix tree") together with binary search trees to compute all pairs of substrings $\boldsymbol{u}$ of $\boldsymbol{x}$ such that

- $\boldsymbol{uvu}$ is a substring of $\boldsymbol{x}$; and
- the ***gap*** $|\boldsymbol{v}| \in r_1..r_2$, where r_1, r_2 are given such that $1 \leq r_1 \leq r_2 \leq n{-}2$.

Their algorithm executes in time $O(n \log n + q)$, where as above q is the number of output pairs $\boldsymbol{u}$; if no upper bound is specified ($r_2 = n-2$), the time reduces to $O(n{+}q)$ plus $O(n \log \sigma)$ suffix tree construction time. Since in bioinformatics applications, $\sigma = 4$, the overall time in practice is $O(n{+}q)$.

P3.2 Also in 2000 Kolpakov & Kucherov [68], using quite different methods, described an $O(n{+}q)$-time algorithm to compute all $\boldsymbol{u}$ such that $\boldsymbol{uvu}$ is a substring of $\boldsymbol{x}$ for some fixed $|\boldsymbol{v}| = r$. They employ the LZ factorization of $\boldsymbol{x}$ [4] together with a modification of the [75] divide-and-conquer all-repetitions approach (see above, Section 5) and KMP pattern-matching [63].

P4.1 Iliopoulos *et al.* [51], Bakalis *et al.* [12], and Antoniou *et al.* [5] use various forms of "generalized" suffix tree (over multiple strings) to extend the gap problem P3 to a set $S = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_k}\}$ of strings, also to the output of complete NE repeats rather than pairs. Making use of appropriate padding with a special character, it may be supposed that each $\boldsymbol{x_j}$, $j = 1, 2, \ldots, k$, is of length n. For each $\boldsymbol{x_j}$ these authors compute $\boldsymbol{uv_1uv_2} \cdots \boldsymbol{v_{m_j-1}u}$, where m_j is the number of occurrences of $\boldsymbol{u}$ accepted in $\boldsymbol{x_j}$. We call this the ***gapped complete NE repeat problem on multiple strings***.

Various constraints may be applied:

- ***gap lengths*** $|\boldsymbol{v_i}|$ may be bounded as in P3;
- ***multiplicities*** m_j of the NE repeat may be required to satisfy a lower bound;
- the number q of strings in which an acceptable repeat occurs may be required to satisfy a lower bound q_0 (no output unless $q \geq q_0$).

P4.2 In [58] Iliopoulos *et al.* apply suffix arrays rather than suffix trees to this problem.

7 Future Challenges

In this section I outline some possible research directions for the future:

Covers & k-Covers

(1) The computation of maximal quasiperiodicities has apparently not been attempted using suffix arrays instead of suffix trees; such an algorithm, if it existed, would no doubt use much less space and perhaps also execute faster.
(2) Furthermore, in view of the results achieved to date in bounding the number of runs and the expected number of runs (see Section 5), it could well be of interest to try to estimate more precisely the number of maximal quasiperiodicities that can occur in any string of given length n. More exact combinatorial knowledge might lead to the design of algorithms that could avoid massive preprocessing (see also below, item (7)).
(3) The restrictions on the λ-covers algorithm are significant; it would be desirable to find an approach that would avoid them, even if the complexity remained at $O(n^2)$. Moreover, it appears that it should be possible to design an algorithm with lower complexity, in view of the fact that the λ-seeds problem can also be handled in time $O(n^2)$.

Seeds

(4) An improvement to λ-covers would presumably have a spillover effect on the λ-seeds algorithm.
(5) It remains an open problem whether the shortest right seed array $\mathrm{RS}_{\mathrm{min}}$ can, like its counterparts, be computed in linear time. Moreover, it appears at least possible that an $o(n^2)$ algorithm exists to compute the shortest seed array $\mathrm{S}_{\mathrm{min}}$.
(6) More generally, to what extent can the cover array of [72] be extended to seeds — that is, giving *all* the seeds of every prefix of $\boldsymbol{x}$?

Repetitions & Runs

(7) It has been established that the maximum number $\rho(n)$ of runs in $\boldsymbol{x}[1..n]$ is relatively small, but the result comes primarily from computation, not from combinatorial knowledge. However, if one could establish that whenever two squares begin at nearby locations, it must as a result follow that at some other location no square begins, then one could formulate an amortization argument that therefore $\rho(n)/n \leq n$. This simple observation has prompted a sequence of research papers over the last few years [32, 94, 69, 36] that has greatly extended previous combinatorial insight into squares in strings ("The Three Squares Lemma" [30]), and that may well have algorithmic consequences. The case that has been considered to date involves two squares $\boldsymbol{u}^2$ and $\boldsymbol{v}^2$, $|\boldsymbol{u}| < |\boldsymbol{v}| < 2|\boldsymbol{u}|$, at the same position, with a third square $\boldsymbol{w}^2$,

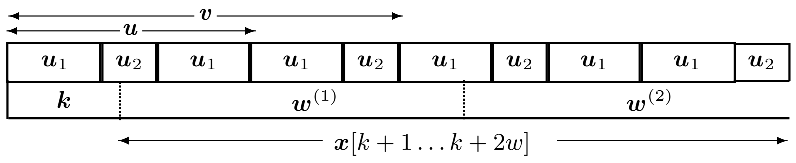

Fig. 2. Overlapping Squares

$|\boldsymbol{v}|-|\boldsymbol{u}| < |\boldsymbol{w}| < |\boldsymbol{v}|$, located k positions to the right, $0 \le k < |\boldsymbol{v}|-|\boldsymbol{u}|$.

Since many subcases need to be considered (Figure 2 shows one of them), the combinatorics are complicated, but what seems to be true is that three such squares cannot exist — more precisely, they can exist only trivially because the string breaks down locally into a repetition of small period that is easily recognized in a left-to-right scan. Of course there are other cases to be considered than the one specified above — and these have not yet been well defined — but the possibility exists that with precise combinatorial knowledge about the existence of squares, an algorithmic approach could be devised that would greatly reduce the time required to compute runs. One man's hobby-horse, perhaps.

Repeats

(8) It appears that the suffix tree/array technology is challenged by the difficulty of the problem P4. There seems to be much scope for new ideas and approaches in this context.

References

[1] Mohamed I. Abouelhoda, Stefan Kurtz, & Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algorithms 2* (2004) 53–86.

[2] Don Adjeroh, Tim Bell & Amar Mukherjee, *The Burrows Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, Springer-Verlag (2008) 351 pp.

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

[4] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, W. F. Smyth, German Tischler & Munina Yusufu, **A comparison of index-based Lempel-Ziv LZ77 factorization algorithms**, *ACM Computing Surveys* (2012) to appear.

[5] Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos & Pierre Peterlongo, **Application of suffix trees for the acquisition of common motifs with gaps in a set of strings**, *Proc. First Internat. Conf. on Language & Automata Theory & Applications* (2007) 57–66.

[6] Alberto Apostolico, **The myriad virtues of suffix trees**, *Combinatorial Algorithms on Words* (NATO ASI Series F12), Springer-Verlag (1985) 85–96.

[7] Alberto Apostolico & Dany Breslauer, *Of Periods, Quasiperiods, Repetitions and Covers*, Tech. Report CSD-TR 97-017, Department of Computer Science, Purdue University (1997) 13 pp.

[8] Alberto Apostolico & Andrzej Ehrenfeucht, *Efficient Detection of Quasiperiodicities in Strings*, Tech. Report No. 49, Computer Science Dept., Purdue University (1990).

[9] Alberto Apostolico & Andrzej Ehrenfeucht, **Efficient detection of quasiperiodicities in strings**, *Theoret. Comput. Sci. 119* (1993) 247–265.

[10] Alberto Apostolico, Martin Farach & Costas S. Iliopoulos, **Optimal superprimitivity testing for strings**, *Inform. Process. Lett. 39* (1991) 17–20.

[11] Alberto Apostolico & Franco P. Preparata, **Optimal off-line detection of repetitions in a string**, *Theoret. Comput. Sci. 22* (1983) 297–315.

[12] A. Bakalis, Costas S. Iliopoulos, S. Sioutas, E. Theodoridis, A. Tsakalidis & K. Tsichlas, **Locating maximal multirepeats in multiple strings under various constraints**, *The Computer Journal 50–2* (2007) 178–185.

[13] Jean Berstel & Luc Boasson, **Partial words and a theorem of Fine and Wilf**, *Theoret. Comput. Sci. 218* (1999) 135–141.

[14] Francine Blanchet-Sadri, *Algorithmic Combinatorics on Partial Words*, Chapman & Hall/CRC (2007) 385 pp.

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

[16] Gerth Stolting Brodal, Rune B. Lyngso, Christian N. S. Pedersen & Jens Stoye, **Finding maximal pairs with bounded gap**, *J. Discrete Algorithms 1* (2000) 77–103.

[17] Gerth Stolting Brodal & Christian N. S. Pedersen, **Finding maximal quasiperiodicities in strings**, *Proc. 11th Annual Symp. Combinatorial Pattern Matching*, Lecture Notes in Computer Science, LNCS 1848, Springer-Verlag (2000) 397–411.

[18] Michael Burrows & David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Tech. Report 124, Digital Equipment Corporation (1994).

[19] Gang Chen, Simon J. Puglisi & W. F. Smyth, **Fast & practical algorithms for computing all the runs in a string**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching*, Bin Ma & Kaizhong Zhang (eds.), Lecture Notes in Computer Science, LNCS 4580, Springer-Verlag (2007) 307–315.

[20] Gang Chen, Simon J. Puglisi & W. F. Smyth, **Lempel-Ziv factorization using less time & space**, *Mathematics in Computer Science 1-4*, Joseph Chan and Maxime Crochemore (eds.) (2008) 605–623.

[21] Michalis Christou, Maxime Crochemore, Ondrej Guth, Costas S. Iliopoulos & Solon P. Pissis, **On the right-seed array of a string**, Proc. 17th Annual International Computing & Combinatorics Conference, Lecture Notes in Computer Science, LNCS 6842, Springer-Verlag (2011) 492–502.

[22] Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder & Tomasz Walen, **Efficient seeds computation revisited**, *Proc. 22nd Annual Symp. Combinatorial Pattern Matching*, Raffele Giancarlo & Giovanni Manzini (eds.), Lecture Notes in Computer Science, LNCS 6661, Springer-Verlag (2011) 350–363.

[23] Richard Cole, Costas S. Iliopoulos, Manal Mohamed, W. F. Smyth & Lu Yang, **The complexity of the minimum k-cover problem**, *J. Automata, Languages & Combinatorics 10–5/6* (2005) 641–653.

[24] Maxime Crochemore, **An optimal algorithm for computing all the repetitions in a word**, *Inform. Process. Lett. 12–5* (1981) 244–248.

[25] Maxime Crochemore & Lucian Ilie, **Computing longest previous factor in linear time and applications**, *Inform. Process. Lett. 106–2* (2008) 75–80.

[26] Maxime Crochemore & Lucian Ilie, **Maximal repetitions in strings**, *J. Comput. Sys. Sci.* (2008) 796–807.

[27] Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter & Tomasz Waleń, **LPF computation revisited**, *Proc. 20th Internat. Workshop on Combinatorial Algs.*, Jiri Fiala, Jan Kratochvil & Mirka Miller (eds.), Lecture Notes in Computer Science, LNCS 5874, Springer-Verlag (2009) 158–169.

[28] Maxime Crochemore, Lucian Ilie, & W. F. Smyth, **A simple algorithm for computing the Lempel-Ziv factorization**, *Proc. 18th Data Compression Conference (DCC'08)*, J. A. Storer & M. W. Marcellin (eds.) (2008) 482–488.

[29] Maxime Crochemore, Lucian Ilie & Liviu Tinta, **Towards a solution to the "runs" conjecture**, *Proc. 19th Annual Symp. Combinatorial Pattern Matching*, P. Ferragina & G. Landau (eds.), Lecture Notes in Computer Science, LNCS 5029, Springer-Verlag (2008) 290–302.

[30] Maxime Crochemore & Wojciech Rytter, **Squares, cubes, and time-space efficient strings searching**, *Algorithmica 13* (1995) 405–425.

[31] L. J. Cummings & W. F. Smyth, **Weak repetitions in strings**, *J. Combinatorial Math. & Combinatorial Computing* 24 (1997) 33–48.

[32] Kangmin Fan, Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A new periodicity lemma**, *SIAM J. Discrete Math. 20–3* (2006) 656–668.

[33] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc. 38th IEEE Symp. Found. Computer Science*, IEEE Computer Society (1997) 137–143.

[34] Gabriele Fici, Thierry Lecroq, Arnaud Lefebvre & Élise Prieur-Gaston, **Computing Abelian periods in words**, *Proc. Prague Stringology Conf.* (2011) 184–196.

[35] N. J. Fine & H. S. Wilf, **Uniqueness theorems for periodic functions**, *Proc. Amer. Math. Soc. 16* (1965) 109–114.

[36] Frantisek Franek, Robert C. G. Fuller, Jamie Simpson & W. F. Smyth, **More results on overlapping squares**, *J. Discrete Algorithms* (2012) to appear.

[37] Frantisek Franek, Jan Holub, W. F. Smyth & Xiangdong Xiao, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[38] Frantisek Franek, Mei Jiang & Chia-Chun Weng, **An improved version of the runs algorithm based on Crochemore's partitioning algorithm**, *Proc. Prague Stringology Conf.*, Department of Theoretical Computer Science, Czech Technical University (2011) 98–105.

[39] Frantisek Franek, Mei Jiang & Chia-Chun Weng
`http://www.cas.mcmaster.ca/~franek/research/crochB7.cpp.txt`

[40] Frantisek Franek, R. J. Simpson & W. F. Smyth, **The maximum number of runs in a string**, *Proc. 14th Australasian Workshop on Combinatorial Algs.*, Mirka Miller & Kunsoo Park (eds.) (2003) 26–35.

[41] Frantisek Franek, W. F. Smyth & Yudong Tang, **Computing all repeats using suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 579–591.

[42] Frantisek Franek, W. F. Smyth & Xiangdong Xiao, **A note on Crochemore's repetitions algorithm — a fast space-efficient approach**, *Nordic J. Computing 10–1* (2003) 21–28.

[43] Edward Fredkin, **Trie memory**, *Commun. Assoc. Comput. Mach. 3-9* (1960) 490–499.

[44] Mathieu Giraud, **Not so many runs in strings**, *Proc. 2nd Internat. Conf. on Language & Automata Theory & Applications*, Carlos Martín-Vide, Friedrich Otto & Henning Fernau (eds.), Lecture Notes in Computer Science, LNCS 5196, Springer-Verlag (2008) 232–239.

[45] R. Groult & G. Richomme, **Optimality of some algorithms to detect quasiperiodicities**, *Theoret. Comput. Sci. 411/34–36* (2010) 34–36.

[46] Qing Guo, Hui Zhang & Costas S. Iliopoulos, **Computing the λ-seeds of a string**, *Proc. 2nd International Conference on Algorithmic Aspects in Information & Management*, Lecture Notes in Computer Science, LNCS 4041, Springer-Verlag (2006) 303–313.

[47] Qing Guo, Hui Zhang & Costas S. Iliopoulos, **Computing the λ-covers of a string**, *Inform. Sciences 177–19* (2007) 3957–3967.

[48] Dan Gusfield, *Algorithms on Strings, Trees & Sequences*, Cambridge University Press (1997) 534 pp.

[49] John E. Hopcroft, **An $n \log n$ algorithm for minimizing states in a finite automaton**, *Theory of Machines and Computations*, Z. Kohavi & A. Paz (eds.), Academic Press (1971).

[50] Lucian Ilie & W. F. Smyth, **Minimum unique substrings and maximum repeats**, *Fundamenta Informaticae 110–1/4* (2011) 183–195.

[51] Costas S. Iliopoulos, James McHugh, Pierre Peterlongo, Nadia Pisanti, Wojciech Rytter & Marie-France Sagot, **A first approach to finding common motifs with gaps**, *Internat. J. Foundations of Computer Science 16–6* (2005) 1145–1155.

[52] Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, W. F. Smyth & Athanasios K. Tsakalidis, **String regularities with don't cares**, *Nordic J. Computing 10–1* (2003) 40–51.

[53] Costas S. Iliopoulos, Manal Mohamed & W. F. Smyth, **New complexity results for the k-covers problem**, *Information Sciences 181* (2011) 2571–2575.

[54] Costas S. Iliopoulos, D. W. G. Moore & Kunsoo Park, **Covering a string**, *Algorithmica 16* (1996) 288–297.

[55] Costas S. Iliopoulos & Laurent Mouchard, **Quasiperiodicity: from detection to normal forms**, *J. Automata, Languages & Combinatorics 4–3* (1999) 213–228.

[56] Costas S. Iliopoulos, Laurent Mouchard & M. Sohel Rahman, **A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching**, *Math. in Computer Science 3–4* (2010) 433–442.

[57] Costas S. Iliopoulos & W. F. Smyth, **On-line algorithms for k-covering**, *Proc. Ninth Australasian Workshop on Combinatorial Algs.*, Curtin University of Technology (1998) 64–73.

[58] Costas S. Iliopoulos, W. F. Smyth & Munina Yusufu, **Faster algorithms for computing maximal multirepeats in multiple sequences**, *Fundamenta Informaticae 97–3*, special StringMasters issue, Ryszard Janicki, Simon J. Puglisi & M. Sohel Rahman (eds.) (2009) 311–320.

[59] Juha Kärkkäinen, Giovanni Manzini & Simon J. Puglisi, **Permuted longest-common-prefix array**, *Proc. 20th Annual Symp. Combinatorial Pattern Matching*, Gregory Kucherov & Esko Ukkonen (eds.), Lecture Notes in Computer Science, LNCS 5577 (2009) 181–192.

[60] Juha Kärkkäinen & Peter Sanders, **Simple linear work suffix array construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, Lecture Notes in Computer Science, LNCS 2719, Springer-Verlag (2003) 943–955.

[61] Toru Kasai, Gunho Lee, Hiroki Akimura, Setsuo Arikawa & Kunsoo Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, Amihood Amir & Gad M. Landau (eds.), Lecture Notes in Computer Science, LNCS 2089, Springer-Verlag (2001) 181–192.

[62] Dong Kyue Kim, Jeong Seop Sim, Heejin Park & Kunsoo Park, **Linear-time construction of suffix arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, Ricardo Baeza-Yates, Edgar Chávez & Maxime Crochemore (eds.), Lecture Notes in Computer Science, LNCS 2676, Springer-Verlag (2003) 186–199.

[63] Donald E. Knuth, James H. Morris & Vaughan R. Pratt, **Fast pattern matching in strings**, *SIAM J. Computing 6–2* (1977) 323–350.

[64] Pang Ko & Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), Lecture Notes in Computer Science, LNCS 2676, Springer-Verlag (2003) 200–210.

[65] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter & Tomasz Waleń, **A linear time algorithm for seeds computation**, *Proc. ACM-SIAM Symp. Discrete Algs.* (2012) 1095–1112.

[66] Roman Kolpakov & Gregory Kucherov, **Finding maximal repetitions in a word in linear time**, *Proc. 40th Annual IEEE Symp. Found. Computer Science* (1999) 596–604.

[67] Roman Kolpakov & Gregory Kucherov, **On maximal repetitions in words**, *J. Discrete Algorithms 1* (2000) 159–186.

[68] Roman Kolpakov & Gregory Kucherov, **Finding repeats with fixed gap**, *Proc. 7th International Symposium on String Processing & Information Retrieval* (2000) 162–168.

[69] Evguenia Kopylova & W. F. Smyth, **The three squares lemma revisited**, *J. Discrete Algorithms 11* (2012) 3–14.

[70] Stefan Kurtz, **Reducing the space requirement of suffix trees**, *Software, Practice & Experience 29–13* (1999) 1149–1171.

[71] Abraham Lempel & Jacob Ziv, **On the complexity of finite sequences**, *IEEE Trans. Information Theory 22* (1976) 75–81.

[72] Yin Li & W. F. Smyth, **Computing the cover array in linear time**, *Algorithmica 32–1* (2002) 95–106.

[73] M. Lothaire, *Combinatorics on Words*, 2nd edition, Cambridge University Press (1997) 238 pp.

[74] Michael G. Main, **Detecting leftmost maximal periodicities**, *Discrete Applied Maths. 25* (1989) 145–153.

[75] Michael G. Main & Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algorithms 5* (1984) 422–432.

[76] Udi Manber & Gene W. Myers, **Suffix array: a new method for on-line string searches**, *Proc. First Annual ACM-SIAM Symp. Discrete Algs.* (1990) 319-327.

[77] Udi Manber & Gene W. Myers, **Suffix array: a new method for on-line string searches**, *SIAM J. Computing 22–5* (1993) 935–948.

[78] G. Manzini, **Two space saving tricks for linear time LCP computation**, *Proc. 9th Scandinavian Workshop on Algorithm Theory*, T. Hagerup & J. Katajainen (eds.), Lecture Notes in Computer Science, LNCS 3111, Springer-Verlag (2004) 372–383.

[79] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai & Ayumi Shinohara, **New lower bounds for the maximum number of runs in a string**, *Proc. Prague Stringology Conf.*, Jan Holub & Jan Zdarek (eds.) (2008) 140–145.

[80] Edward M. McCreight, **A space-economical suffix tree construction algorithm**, *J. Assoc. Comput. Mach. 32–2* (1976) 262–272.

[81] Dennis Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *Inform. Process. Lett. 50–5* (1994) 239–246.

[82] Dennis Moore & W. F. Smyth, **A correction to "An optimal algorithm to compute all the covers of a string"**, *Inform. Process. Lett. 54* (1995) 101–103.

[83] Yuta Mori, **libdivsufsort**: `http://code.google.com/p/libdivsufsort/`

[84] Kaziyuki Narisawa, Shunsuke Inenaga, Hideo Bannai & Masayuki Takeda, **Efficient computation of substring equivalence classes with suffix arrays**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching*, Bin Ma

& Kaizhong Zhang (eds.), Lecture Notes in Computer Science, LNCS 4580, Springer-Verlag (2007) 340–351.

[85] Ge Nong, Sen Zhang & Wai Hong Chan, **Linear time suffix array construction using D-critical substrings**, *Proc. 20th Annual Symp. Combinatorial Pattern Matching*, Gregory Kucherov & Esko Ukkonen (eds.), Lecture Notes in Computer Science, LNCS 5577, Springer-Verlag (2009) 54–67.

[86] Daisuke Okanohara & Kunihiko Sadakane, **An online algorithm for finding the longest previous factors**, *Proc. 16th Annual European Symp. on Algs.*, D. Halperin & K. Mehlhorn (eds.), Lecture Notes in Computer Science, LNCS 5193, Springer-Verlag (2008) 595–707.

[87] Simon J. Puglisi & R. J. Simpson, **The expected number of runs in a word**, *Australasian J. Combinatorics 42* (2008) 45–54.

[88] Simon J. Puglisi, R. J. Simpson & W. F. Smyth, **How many runs can a string contain?**, *Theoret. Comput. Sci. 401* (2008) 165–171.

[89] Simon J. Puglisi, W. F. Smyth & Munina Yusufu, **Fast optimal algorithms for computing all the repeats in a string**, *Proc. Prague Stringology Conf.*, Jan Holub & Jan Zdarek (eds.) (2008) 161–169.

[90] Simon J. Puglisi, W. F. Smyth & Munina Yusufu, **Fast optimal algorithms for computing all the repeats in a string**, *Math. in Computer Science 3–4*, special issue "Advances in Combinatorial Algorithms II", Manolis Christodoulakis & Costas S. Iliopoulos (eds.) (2010) 373–389.

[91] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys 39–2* (2007) Article 4, 1–31.

[92] Simon J. Puglisi & Andrew Turpin, **Space-time tradeoffs for longest-common-prefix array computation**, *Proc. 19th Internat. Symp. Algs. & Computation*, S.-H. Hong, H. Nagamochi & T. Fukunaga (eds.) (2008) 124–135.

[93] Wojciech Rytter, **The number of runs in a string: improved analysis of the linear upper bound**, *Proc.* 23rd *Symp. Theoretical Aspects of Computer Science*, B. Durand & W. Thomas (eds.), LNCS 2884, Springer-Verlag (2006) 184–195.

[94] R. J. Simpson, **Intersecting periodic words**, *Theoret. Comput. Sci. 374* (2007) 58–65.

[95] Jamie Simpson, **Modified Padovan words and the maximum number of runs in a word**, *Australasian J. Combinatorics 46* (2010) 129–145.

[96] W. F. Smyth, **Repetitive perhaps, but certainly not boring**, *Theoret. Comput. Sci. 249–2* (2000) 343–355.

[97] Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[98] W. F. Smyth & Shu Wang, **A new approach to the periodicity lemma on strings with holes**, *Theoret. Comput. Sci. 410–43*, Special Issue for Maxime Crochemore's 60th birthay, Costas S. Iliopoulos & Wojciech Rytter (eds.) (2009) 4295–4302.

[99] Axel Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7* (1906) 1–22.

[100] Esko Ukkonen, **On-line construction of suffix trees**, *Algorithmica 14* (1995) 249–260.

[101] Peter Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching & Automata Theory* (1973) 1–11.

[102] Jacob Ziv & Abraham Lempel, **A universal algorithm for sequential data compression**, *IEEE Trans. Information Theory 23* (1977) 337–343.