

## String Covering: A Survey

**Neerja Mhaskar\***

*Department of Computing and Software*

*McMaster University*

*1280 Main Street West, Hamilton, ON L8S 4L8, Canada*

*pophlin@mcmaster.ca*

**W.F. Smyth<sup>†</sup>**

*Department of Computing and Software*

*McMaster University*

*1280 Main Street West, Hamilton, ON L8S 4L8, Canada*

*smyth@mcmaster.ca*

---

**Abstract.** The study of strings is an important combinatorial field that precedes the digital computer. Strings can be very long, trillions of letters, so it is important to find compact representations. Here we first survey various forms of one potential compaction methodology, the *cover* of a given string  $x$ , initially proposed in a simple form in 1990, but increasingly of interest as more sophisticated variants have been discovered. We then consider covering by a *seed*; that is, a cover of a *superstring* of  $x$ . We conclude with many proposals for research directions that could make significant contributions to string processing in future.

**Keywords:** strings, quasiperiodicity, covers, seeds

---

\*Address for correspondence: Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, ON L8S 4L8, Canada.

<sup>†</sup>This research was funded by NSERC [Grant No. 10536797].

## 1. Introduction

A *string* (as computer scientists call it) or *word* (as mathematicians call it) is just a sequence  $x = x[1..n]$  of  $n \geq 0$  entries drawn from a finite set of *letters* called an *alphabet*, usually denoted by  $\Sigma$ . Stringology — or combinatorics on words — has existed as a field of scientific enquiry for more than a century, dating back to Axel Thue’s foundational paper in 1906 [1]. Thue showed that infinitely long strings could be constructed, even on a small alphabet, so as to avoid certain simple patterns. This suggests that it could be difficult, at least in some cases, to extract meaning (represented for example by recurring patterns) from a given text. Indeed, from its inception, stringology has been deeply concerned with the presence or absence of *patterns* in strings [2, 3].

In particular, the recognition [4] that the DNA of all organisms is to a first approximation a string on an alphabet  $\Sigma = \{a, c, g, t\}$  has over the last 70 years led to a huge and exponentially increasing bioinformatics literature that deals essentially with the application of stringology to fundamental biological research — the search for biologically significant patterns.

In this survey we follow the study of one of these patterns — the “cover”  $u$  of a string  $x$  — from its initial definition [5] as a “quasiperiodicity” (every position in  $x$  lies within an occurrence of  $u$ ) to more flexible concepts, 30 years later, defined in terms of a repeating substring  $u$  whose occurrences maximize, over all repeating substrings of  $x$ , the positions covered — and which, moreover, can be efficiently computed. More generally, we also discuss methods to find a “seed” of  $x$  — that is, a repeating substring  $u$  of  $x$  that covers a string  $w$  which contains  $x$  as a substring [6, 7] — a more general computation that nevertheless, can often be competitive in terms of execution time. Thus, potentially, the covers/seeds of a string  $x$  could provide a comprehensive, compact and usefully organized representation of  $x$  that would assist in its interpretation and processing.

For those not familiar with the terminology of strings, Section 2 provides basic definitions. Then Section 3 gives an overview of the advances (and occasional retreats) in the various extensions of the idea of a “cover” and their application to some kind of canonical representation of a string. Section 4 goes on to deal with the often non-trivial generalization of covering algorithms to seeds. Finally, Section 5 outlines future research directions that may lead to more useful and precise characterizations of the representative “patterns” contained in a given text.

## 2. Preliminaries

We consider strings  $x = x[1..n]$ , where each entry  $x[i]$  consists of one or more distinct symbols (*letters*) chosen from a set  $\Sigma = \{\lambda_1, \lambda_2, \dots, \lambda_\sigma\}$  of finite size  $\sigma = |\Sigma|$ , called an *alphabet*. It is often convenient to suppose that the elements of  $\Sigma$  are ordered. Every entry  $x[i]$  that consists of a single letter from  $\Sigma$  is said to be *regular*; if every position in  $x$  is regular, then so is  $x$ . However, if any  $x[i]$  consists of a set of two or more letters from  $\Sigma$ , such as  $\{a, c, t\}$  or  $\{1, 3\}$ , then both  $x[i]$  and  $x$  are said to be *indeterminate*. A *partial word* is a commonly-occurring special case of an indeterminate string whose entries are either single letters or  $\Sigma$  itself (usually termed a *hole* and denoted as  $*$ ). Strings whose indeterminate entries also specify a likelihood that each symbol will occur are called *weighted* — these typically occur in bioinformatics applications. For example, a single letter  $\{a, 30; c, 40; t, 30\}$  would indicate that  $a/c/t$  is expected to occur 30/40/30 % of the time, respectively.

Thus any English-language text is a regular string on some  $\Sigma$  consisting of 52 upper and lower case letters, 10 numeric digits, space, and a variety of special symbols — and so, for English,  $\sigma$  may be as much as 90. On the other hand, an indeterminate string could be a DNA fragment such as  $x = \{a, c\}g\{g, t\}c$  on alphabet  $\Sigma = \{a, c, g, t\}$ , some of whose entries ( $\{a, c\}$ ,  $\{g, t\}$ ) are not well defined or somehow optional. The results presented in this survey, unless otherwise indicated, are restricted to regular strings.

The **length** of  $x[1..n]$  is  $|x| = n$ . Two regular strings  $x$  and  $y$  of length  $n$  are said to **match** if  $x[i] = y[i]$ ,  $1 \leq i \leq n$ . The **empty string**  $\varepsilon$  is a string of length  $n = 0$ . A string  $x[i..j]$  is a **substring** (or **factor**) of  $x[1..n]$  if  $1 \leq i \leq j \leq n$ , a **proper substring** if moreover  $j - i + 1 < n$ ; when  $x$  is regular,  $x[i..j]$  is said to be a **repeating substring** if it matches another substring of  $x$  — that is, if there exists  $i' \neq i$  and  $j' \leq n$  such that  $x[i'..j'] = x[i..j]$ . A repeating substring  $u$  is said to be **extendible** if all occurrences of  $u$  in  $x$  are followed or preceded by the same letter; otherwise, **nonextendible**. If  $x$  is a proper substring of a string  $w$ , then we say that  $w$  is a **superstring** of  $x$ . A **prefix** (**suffix**) of nonempty  $x$  is a substring  $x[i..j]$ , where  $i = 1$  ( $j = n$ ) — it is moreover convenient to suppose that every nonempty  $x$  has the empty string as both prefix and suffix. The suffix starting at position  $i$  is sometimes called **suffix  $i$** . If  $x = uv$  for nonempty  $u, v$ , then  $x' = vu$  is said to be a **rotation** of  $x$ .

Given  $x = x[1..n]$  on alphabet  $\Sigma$ , the **Parikh vector**  $P = P_x[1..\sigma]$  of  $x$  is an integer array such that, for  $1 \leq \ell \leq \sigma$ ,  $P[\ell]$  is the number of occurrences of letter  $\lambda_\ell$  in  $x$ . A string  $x$  consisting of  $r \geq 1$  concatenated copies of a string  $u$  is denoted by  $u^r$ . Then  $x$  is said to be **periodic** with **period**  $|u|$  if it can be represented as  $x = u^r u'$ , where  $u'$  is a (possibly empty) prefix of  $u$ , and either  $r \geq 2$  and or else  $r = 1$  and  $u'$  is not empty. In the former case ( $r \geq 2$ ),  $u^r$  is said to be a **repetition**; any string that is not a repetition is said to be **primitive**. A **border** of  $x$  is a proper prefix  $u'$  of  $x$  that matches a suffix of  $x$ ; thus every nonempty string has an empty border. Note that if  $x$  has a nonempty border  $u'$ , then  $x$  is necessarily periodic with period  $|x| - |u'|$ . Furthermore, if  $u'$  is a border of  $x$ , then every border of  $u'$  is also a border of  $x$ . For example, the string  $x = ababaaba$  has borders  $aba$ ,  $a$  and  $\varepsilon$ , hence periods  $8 - 3 = 5$  and  $8 - 1 = 7$ ; and note that the shorter border  $a$  must also be a border of the longer border  $aba$ .

The **frequency**  $f_{x,u}$  of a substring  $u$  in a string  $x$  is the number of occurrences of  $u$  in  $x$ . In the preceding example, choosing  $u$  to be the border  $aba$ , we see that  $f_{ababaaba,aba} = 3$ . In fact, we observe that *every* position in  $x$  lies within an occurrence of  $u = aba$ . We formalize this idea with the definition of a **cover** of  $x$ ; that is, a repeating substring  $u$  in  $x$  such that every position in  $x$  lies within an occurrence of  $u$ . In such a case we say that  $x$  is **quasiperiodic**. From the example  $x = ababababa$ , we discover that a string may have more than one cover — in this case,  $u_1 = aba$ ,  $u_2 = ababa$ ,  $u_3 = abababa$  — such that, moreover, every longer cover  $u_i$  is covered by every shorter one  $u_{i-1}$ ,  $u_{i-2}$ ,  $\dots$ ,  $u_1$ . It is easy to see that every cover of  $x$  must also be a border of  $x$ ; thus the set of covers is a subset of the set of borders, and so contains at most  $\mathcal{O}(n)$  elements. The maximum is achieved by  $x = a^n$ , which has  $n - 1$  covers  $a^i$ ,  $1 \leq i \leq n - 1$ ; more generally, every repetition  $x = u^k$  has  $k - 1$  covers  $u^i$ ,  $1 \leq i < k$ .

As noted in the Introduction, it turns out to be useful to generalize the idea of a cover to that of a **seed**; that is, a proper substring of  $x$  which is a cover of a superstring  $w$  of  $x$  [6, 7]. Then, for

<sup>1</sup>When at least one of  $x, y$  is indeterminate, we require for a match only that the intersection  $x[i] \cap y[i]$  be nonempty.

example, the substring  $u_1 = aba$  is a seed of  $x' = abababab$  since, as we have just seen, it is a cover of  $x = ababababa = x'a$ , a superstring of  $x'$ . An important difference between covers and seeds is that the number of seeds of  $x$  may be  $\Theta(n^2)$ : [8] gives the example  $x = a^m b a^m b a^m b a^m$  of length  $n = 4m + 3$ , whose seeds include the  $\Theta(m^2)$  distinct substrings  $a^i b a^j$  determined by the rule that  $i$  and  $j$  assume all values  $0 \leq i, j \leq m$  such that  $i + j \geq m$ .

In Section 3 we discuss the various (and successively more sophisticated) kinds of cover for strings  $x$  that have been proposed over the last 30 years and outline the methodology for their computation. In Section 4 we then go on to discuss the (often surprisingly different) methods proposed for computing seeds. Many of these methods depend heavily on the data structures surveyed below.

We first explain an important extension of the idea of periodicity. A nonempty substring  $v = x[i..j] = u^r u'$  of  $x$  is said to be a **run of period**  $p = |u|$  in  $x$  if  $r \geq 2$ ,  $u'$  is a possibly empty proper prefix of  $u$ , and there exist *no* integers  $i' \leq i, j' \geq j$  such that

- (1)  $x[i..j]$  is a proper substring of  $x[i'..j']$ ; and
- (2)  $x[i'..j']$  has period  $p$ .

Thus a run, denoted  $(i, j, p)$ , is **maximal**: any extension left or right of  $v$  in  $x$  yields a substring that is *not* a run of period  $p$ .

For example, the string  $x = abacababacabacaba$  contains runs  $x[5..9] = ababa = (ab)^2 a$  of period 2,  $x[7..17] = abacabacaba = (abac)^2 aba$  of period 4, and  $x[1..13] = abacababacaba = (abacab)^2 a$  of period 6. These runs represented as triples are  $r_1 = (5, 9, 2)$ ,  $r_2 = (7, 17, 4)$  and  $r_3 = (1, 13, 6)$ , respectively. Note however that  $x[7..16] = (abac)^2 ab$  and  $x[10..17] = (caba)^2$  are *not* runs because they are not maximal.

A **suffix array**  $\mathcal{SA}_x$  of  $x$  [9, 10, 11, 12] is an integer array of length  $n$ , where  $\mathcal{SA}_x[i]$  is the starting position of the  $i$ -th lexicographically least (lexleast) suffix in  $x$ . (Thus an ordering of the alphabet is required.) The **longest common prefix array**  $\mathcal{LCP}_x$  of  $x$  [9, 10, 13] is an integer array of length  $n$ , where  $\mathcal{LCP}_x[1] = 0$  and  $\mathcal{LCP}_x[i]$ ,  $1 < i \leq n$ , is the length of the longest common prefix of suffixes starting at positions  $\mathcal{SA}_x[i - 1]$  and  $\mathcal{SA}_x[i]$ . Over the last quarter-century these data structures, both of them now recognized [13, 11, 12, 14] to be efficiently computable in  $\mathcal{O}(n)$  time, have come to be used extensively in a wide variety of string algorithms: taken together they permit equal substrings of  $x$  to be identified (since those substrings will naturally occur close together in the sorted suffix array). For example, in Figure 1 the three occurrences of substring  $aba$  in  $x = abaababa$  — starting at positions 1, 4 and 6 — occur conveniently close together in  $\mathcal{SA} = \mathcal{SA}_x$ , identified by  $\mathcal{SA}[3..6] = 614$ . Furthermore, since  $\mathcal{LCP} = \mathcal{LCP}_x$  is keyed to  $\mathcal{SA}$ , the value  $\mathcal{LCP}[4] = 3$  tells us that there are equal substrings of length 3 beginning at  $x[\mathcal{SA}[3]] = x[6]$  and  $x[\mathcal{SA}[4]] = x[1]$ . Thus these two integer arrays of length  $n$  are powerful computational tools for the analysis of strings — for, in a sense, determining their “meaning”.

The  $\mathcal{SA}/\mathcal{LCP}$  arrays provide compact storage for a more general structure, called the **suffix tree** of  $x$ , denoted by  $\mathcal{ST} = \mathcal{ST}_x$ . As shown in Figure 1,  $\mathcal{ST}$  is a search tree with  $n$  leaf nodes representing the  $n$  nonempty suffixes of  $x$ ; each edge descending from each internal node represents a distinct letter or substring, which are available in ascending lexicographic order. For example, the edges descending

	1	2	3	4	5	6	7	8
$x =$	a	b	a	a	b	a	b	a
$SA_x =$	8	3	6	1	4	7	2	5
$LCP_x =$	0	1	1	3	3	0	2	2
$RSF_x =$	0	5	5	3	3	0	3	3
$OLP_x =$	0	0	0	1	0	0	0	0

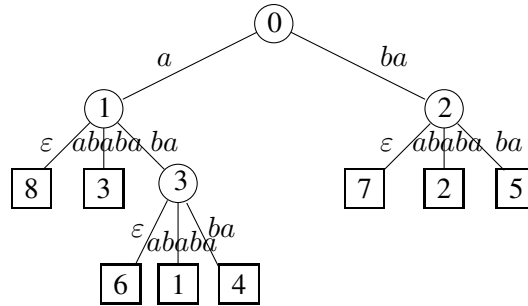


Figure 1: Suffix array,  $LCP/RSF/OLP$  arrays and corresponding suffix tree for  $x = abaababa$  — adapted from [14].

from node 0 in Figure 1 correspond to  $a$  and  $ba$ , the only possible prefixes of any suffix of  $x$ . At node 1, the only possible continuations from  $a$  are  $\epsilon$ , corresponding to position 8 in  $x$ ,  $ababa$ , corresponding to position 3, and  $ba$ , corresponding to one of positions 6, 1, 4. Thus a preorder traversal of  $ST_x$  yields the suffixes of  $x$  in ascending lexorder (exactly the entries referenced by  $SA_x$ ), while the path from the root of the tree to each node spells out the common prefix to all of that node’s descendants in  $ST_x$  (exactly the entries in  $LCP_x$ ).

Like  $SA_x$  and  $LCP_x$ ,  $ST_x$  can be computed in  $\mathcal{O}(n)$  time [2, Ch. 5.2], but nevertheless, as a practical matter, its construction is much slower. Moreover, the tree structure, due to the requirement for pointers and other auxiliary information, normally consumes much more space in computer memory. However, suffix trees are still used in many applications because the basic tree can be “annotated” in various ways with information useful for processing. See however [15].

The suffix tree was first proposed and computed almost half a century ago [16], the suffix array 30 years ago [9]. Surveys of their computation and use are available in [14, 17, 18].

For definitions of the  $RSF_x$  (Repeating Substring Frequency) and  $OLP_x$  (Overlapping Positions) arrays, see Subsection 3.4.

### 3. Covers in strings

Before discussing the various forms of cover of a string, we pause to say a little more about the border and the period. Given  $x[1..n]$ , the array  $\beta_x[1..n]$  is said to be the **border array** of  $x$  if  $\beta_x[i]$  is the length of the longest border of  $x[1..i]$ ,  $1 \leq i \leq n$ . Thus for the example  $x = abaababa$ ,  $\beta_x = 00123123$ . We see that in this case  $\beta_x$  provides quite a bit of information about  $x$  — in

fact, with a bit of logic, we can determine the exact structure of  $x$  from 00123123, which we could characterize using two arbitrary letters  $\lambda_1$  and  $\lambda_2$ :  $\lambda_1\lambda_2\lambda_1\lambda_2\lambda_1\lambda_1\lambda_2\lambda_1$ . Similarly, the **period array**  $P_x$  of  $x$  [19] is an integer array where  $P_x[i]$  is the length of the shortest period of  $x[1..i]$ , and its dual the **suffix period array**  $P_{x,suf}$  is defined such that  $P_{x,suf}[i]$  is the length of the shortest period of  $x[i..n]$ . For the above example  $x = ababaaba$ ,  $P_x = 12222555$ ,  $P_{x,suf} = 55333221$ .

Furthermore,  $\beta$  can be elegantly computed in  $\Theta(n)$  time using a famous early (1974) string algorithm [20], giving rise to the tantalizing idea that strings might somehow be organized into useful equivalence classes according to some structural analysis. Of course, a little reflection tells us that, alas, the border array will not satisfy this requirement: 00000000 is the border array of *cabaababa*, *caaaaaaaaa*, *abcdefghi*, and many other very diverse strings! In fact, as shown in [21], the expected maximum length of the border of a string on a binary alphabet is only 1.64 letters — and this value of course decreases precipitously as alphabet size increases.

Closely related to the border array is the **prefix array** of  $x$ ; that is, an array  $\pi = \pi[1..n]$ , where  $\pi_x[i]$  is the length of the longest substring at position  $i$  of  $x$  that matches a prefix of  $x$ . Thus, in the above example  $x = ababaaba$ ,  $\pi_x = 80301301$ . It was shown in [22] that, on regular strings, the border array and the prefix array are equivalent in the sense that each can be computed from the other. However, on indeterminate strings, the prefix array retains its properties — in particular, identifying all the borders of every prefix — whereas the border array does not [23].

A recent paper [24] by Iliopoulos & Radoszewski discusses subquadratic solutions to compute both the quantum and deterministic border arrays (see Section 3.9) and the prefix array of partial and indeterminate strings. They describe  $\mathcal{O}(n\sqrt{n\log n})$ -time algorithms to compute these arrays on partial words. Then they propose  $\mathcal{O}(n\sqrt{n})$ -time algorithms to compute the prefix array and the quantum border array of an indeterminate string over a constant-sized alphabet. They also go on to show that, provided the Strong Exponential Time Hypothesis (SETH) holds, no efficient algorithms exist to compute both the quantum and deterministic border arrays, and prefix array of an arbitrary indeterminate string over a general alphabet. As we discover below (Section 3.9), the cover array computation is even more restricted. These results have recently been put to use in [25, 26], where the prefix array, rather than the border array, has been introduced into the Knuth-Morris-Pratt (KMP, [27]) and Boyer-Moore (BM, [28]) pattern-matching algorithms, in order to do efficient pattern-matching on indeterminate strings: for text of length  $n$ , pattern of length  $m$ , both indeterminate, KMP and BM execute in  $\mathcal{O}(m\sqrt{mn})$  time.

### 3.1. Covers & cover arrays

In 1990 Apostolico and Ehrenfeucht [5, 29] introduced the idea of a cover of a string, defined in Section 2, as a means of providing a description of some strings, at once more succinct and more expressive: *aba* as a kind of abbreviation of *ababaaba*. They described an  $\mathcal{O}(n\log^2 n)$  algorithm to compute all the maximal quasiperiodic substrings of  $x$ , in particular  $x$  itself if quasiperiodic, a result later improved by Iliopoulos and Mouchard [30], also Brodal and Pedersen [31], to  $\mathcal{O}(n\log n)$ . In 1991 Apostolico, Farach and Iliopoulos [32] described a linear-time algorithm to determine the shortest cover of quasiperiodic  $x$ . Then Breslauer [33] published an on-line linear-time algorithm to compute the minimum cover of each prefix of  $x$ , while Moore and Smyth [34, 35] described a

linear-time algorithm to compute *all* the covers of  $x$  itself. A recent paper [36] compares the runtime of these latter three algorithms (AFI [32], B [33], MS [34, 35]), along with implementations of several other generalizations of cover algorithms. Further, in [37] Iliopoulos and Park presented an  $\mathcal{O}(\log \log n)$ -time parallel algorithm to compute all the covers of  $x$ .

Finally, Li and Smyth [38] published an on-line linear-time algorithm to compute the **cover array**  $\gamma_x[1..n]$  of  $x$ , specifying the longest cover of each prefix  $x[1..i]$  of  $x$ , zero for no cover. Since for every cover  $u$  of  $x$ , any cover of  $u$  is also a cover of  $x$ , the cover array  $\gamma$  turns out to be an exact analogy to the border array  $\beta$ , specifying *all* the covers of every prefix of  $x$ :

	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	$a$	$b$	$a$	$a$	$b$	$a$	$b$	$a$	$a$	$b$	$a$	$a$	$b$
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	1	5
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5

Thus the cover array, even more precisely than the border array, can describe the structure of certain strings  $x$ .

This observation suggests the possibility of inferring a string that corresponds to a given cover array, first stated and solved in [39]: given a valid cover array  $\gamma$  of length  $n$ , find in linear time a corresponding string  $x$  on an alphabet of minimum size whose cover array is  $\gamma$ . Then three years later, in [40, 41], a remarkable linear-time algorithm was described that, for every valid  $\gamma$ , computes a corresponding string on a two-letter alphabet.

In this context, a recent paper [42] shows that  $x$ , though not necessarily on a minimum alphabet, can be determined in linear time from different kinds of repetitions or symmetries present in the string — in particular, corresponding to a given valid border array, prefix array, or other features.

Unfortunately, far fewer strings have covers than have nonempty borders [21], so that the range of application of the cover array is correspondingly limited. Even short strings on a small alphabet rarely have a cover — for  $n = 4$  and  $\sigma = 2$ , only four of 16 distinct nonempty strings have a cover — so that for longer strings on a larger alphabet, almost always  $\gamma_x[i] = 0$ .

Nevertheless, almost 20 years after the publication of the cover array algorithm, two recent papers have refocussed attention on cover computation by describing algorithms to compute covers, first of every rotation, then of every substring, of a given string  $x$ :

- In [43, 44] Crochemore *et al.* first describe an  $\mathcal{O}(n \log n)$ -time algorithm to compute the shortest cover of every rotation of  $x$ , then an  $\mathcal{O}(n)$ -time algorithm to compute the shortest among these covers. This result is further improved by the same authors in [45] where they propose an  $\mathcal{O}(n)$ -time algorithm for the same problem.
- In [46] the same authors preprocess  $x$  in  $\mathcal{O}(n \log n)$  time and space in order to be able to compute, for any selected substring  $u = x[i..j]$  of  $x$ , both the shortest cover (in time  $\mathcal{O}(\log n \log \log n)$ ) and in addition *all* the covers (in time  $\mathcal{O}(\log n (\log \log n)^2)$ ).

These algorithms are complex, requiring not only computation of  $STx$ , but also of all the seeds of  $x$ . For this latter calculation, they make use of a linear-time algorithm [8], discussed in Section 4, that

computes a linear encoding (a **package representation**) of the seeds — even though, as we have seen, their occurrences may be quadratic in number.

Another recent development has also inspired renewed interest in forms of quasiperiodicity: the generalization of pattern matching and periodicity under a **Substring Consistent Equivalence Relation** (SCER) denoted by  $\approx$ . For strings  $\mathbf{x}$  and  $\mathbf{y}$ , [47] defines  $\mathbf{x} \approx \mathbf{y}$  iff  $|\mathbf{x}| = |\mathbf{y}|$  and  $\mathbf{x}[i..j] \approx \mathbf{y}[i..j]$  for every  $1 \leq i \leq j \leq |\mathbf{x}|$ . The authors then describe efficient pattern-matching using  $\approx$  as well as an analogue of Fine & Wilf’s periodicity lemma [48]. This work is followed up in [49], where the cover array is defined for SCERs and existing shortest/longest cover array algorithms are appropriately generalized.

On the other hand, another recent theoretical result provides a basis for understanding that quasiperiodic strings must be in some sense rare: Amir *et al.* [50] show that any two distinct quasiperiodic strings of the same length must differ at more than one position (Hamming distance greater than one).

No doubt due to the scarcity of exact covers in strings, research in this area has for the last quarter century focused on various forms of multiple or approximate cover, as discussed below.

### 3.2. $k/\lambda$ -covers

In 1998 Iliopoulos and Smyth [51] introduced the  $k$ -cover problem: for given  $\mathbf{x}$  and  $k > 1$ , determine a covering set  $U_{t,k} = \{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_t\}$  of  $t$  substrings of  $\mathbf{x}$ , each of length  $k$ , such that every position of  $\mathbf{x}$  lies within some element of  $U_{t,k}$ . For given  $k$ , let  $\tau_k$  denote the minimum value of  $t$  (if it exists) for which  $U_{t,k}$  is a covering set; in this case  $U_{\tau_k,k}$  is said to be a **minimum  $k$ -cover** of  $\mathbf{x}$ . For example, given  $\mathbf{x} = abaabbaaab$  and  $k = 2$ ,  $U_{3,2} = \{ab, aa, bb\}$  and  $\{ab, ba, aa\}$  are minimum 2-covers of  $\mathbf{x}$  with  $\tau_2 = 3$ , while  $U_{3,3} = \{aba, aab, baa\}$  and  $\{aba, aab, bba\}$  are minimum 3-covers with  $\tau_3 = 3$ . Observe that every  $k$ -cover must include the suffix and the prefix of length  $k$ ; thus, whenever  $\mathbf{x}$  has no border of length  $k$ ,  $\tau_k \geq 2$  (if it exists). Of course of particular interest is the smallest value of  $k$  that yields a minimum  $k$ -cover corresponding to the smallest  $\tau_k$ . Thus in the above example, we would prefer  $U_{3,2}$  to  $U_{3,3}$ .

Also in [51] it was “proved” that, for given  $\mathbf{x}$ , the minimum  $\tau_k$  could be computed in polynomial time, a result later found to be incorrect: in 2005 Cole *et al.* [52] showed that the problem of determining the cardinality  $\tau_k$  of a minimum  $k$ -cover is NP-complete for every  $k \geq 2$ . Nevertheless, in the same paper [52], the authors described two  $\mathcal{O}(n \log n)$  algorithms that approximated the minimum  $k$ -cover  $U_{\tau_k,k}$  to within a logarithmic ( $\log n$ ) factor. Then in 2011 (Iliopoulos *et al.* [53]) presented a polynomial-time algorithm to approximate  $U_{\tau_k,k}$  to within a factor  $k$ . For more on this topic see below, Section 3.6.

In [54, 55], summarized in [56], Guo *et al.* introduced the  $\lambda$ -covers problem, a parameterized version of  $k$ -covers: for given positive integers  $k$  and  $\lambda$ , find if possible a set  $S = S_{k,\lambda}$  containing exactly  $\lambda$  substrings of  $\mathbf{x}$ , each of length  $k$ , such that the entries in  $S$  cover  $\mathbf{x}$ . For an alphabet of size  $\sigma$ , the authors presented an  $\mathcal{O}(\sigma k n^2)$  algorithm that solved the problem for values  $1, 2, \dots, k$  and fixed  $\lambda$ . To facilitate an efficient solution, rather than suffix trees, the authors made use of the Equivalence Class and Reversed Equivalence Class Trees (ECT, RECT), introduced in [57], to compute  $S_{k,\lambda}$  whenever possible, still in  $\mathcal{O}(n^2)$  time. However, unfortunately, these results turn out to be incorrect, to an extent that has not yet been made fully precise: in [36, p. 26] Czajka & Radoszewski describe a



counterexample which makes clear that the  $\lambda$ -covers, as defined in these papers, cannot be computed in  $\mathcal{O}(n^2)$  time.

In [58] Radoszewski & Straszyński consider the special case  $\lambda = 2$  of parameterized  $k$ -covers: they describe an algorithm requiring  $\mathcal{O}(n \log^{1+o(1)} n)$  time that, over all  $k$ , specifies all the *pairs* of strings of length  $k$  that cover  $x[1..n]$ . Their algorithm generalizes to  $\lambda > 2$ , but with an order  $n$  multiplicative increase in complexity for each increase in  $\lambda$ .

### 3.3. $\alpha$ -partial cover

In [59] Kociumaka *et al.* introduce the **cover index**  $C = C(x, u)$  of a string  $u$  within  $x$ ; that is, the number of positions in  $x$  covered by a repeating substring  $u$ , which is therefore called a **partial cover** of  $x$ . For example, given  $x = abaababaabaab$  of length  $n = 13$ ,  $C(x, aba) = 11$  and  $C(x, ab) = 10$  are the cover indices for partial covers  $aba$  and  $ab$ , respectively. For a given integer  $\alpha \in 1..n$ , the authors describe an algorithm, executing in  $\mathcal{O}(n \log n)$  time and  $\Theta(n)$  space, to compute all shortest substrings  $u$  of  $x$  such that  $C(x, u) \geq \alpha$ . Thus for the same example  $x$  and  $\alpha = 10$ , the  **$\alpha$ -partial-cover** algorithm would return  $ab$ , the shortest substring covering at least 10 positions. For  $\alpha = 11$ , it would return  $aba$ . Only for the choice  $\alpha = 15$  would it return the full cover  $abaab$  of  $x$ .

However, to correct this deficiency, [59] goes on to describe the **all-partial-covers** algorithm, with the same asymptotic time and space requirements, which computes the  $\alpha$ -partial cover for *all*  $\alpha \in 1..n$ .

This approach was the first to enable computation of a cover of  $x$  that is in some sense optimal without requiring that it also be a border. The price paid for this achievement is that the algorithm requires the use of the *cover suffix tree* – a suffix tree augmented with additional nodes and additional values at each node – thus increasing both time and space requirements. This algorithm, among others, was implemented in [36]. More recently, Radoszewski in [60] proposes a remarkable linear time construction of the cover suffix tree data structure – thus showing that the all-partial-covers of a given string can be computed in  $\mathcal{O}(n)$ -time.

In [61] Flouri *et al.* introduced a restricted version of the partial cover called the **enhanced cover**: that is, a border (rather than a repeating substring) of  $x$  that, over all borders, covers a maximum number of positions in  $x$ . The authors showed that the enhanced cover could be computed in  $\Theta(n)$  time on regular strings. Alatabbi *et al.* [21] extended this idea to the **minimum enhanced cover** (MEC); that is, the shortest border yielding maximum coverage. For example, the borders  $aba$  and  $ababa$  are both enhanced covers of  $x = ababaaabababa$ , covering all but one position, but  $aba$  is the MEC. In [21] an algorithm was proposed that, according to tests, performed somewhat faster in practice, and that moreover, by using the prefix array rather than the border array for computation, extended the cover calculation to indeterminate strings — resulting in an average-case  $\mathcal{O}(n \log n)$ , worst-case  $\mathcal{O}(n^2)$  algorithm. See also Section 4.1 for a discussion of the **enhanced seed**.

### 3.4. Maximal cover

Let  $M = M_x$  denote the maximum number of positions in  $x$  covered by *any* repeating substring of  $x$ . Similar to the  $\alpha$ -partial cover described in Section 3.3, [62] computes a **maximal cover**<sup>2</sup>  $u_M$ ,

---

<sup>2</sup>There called “optimal” cover.

a repeating substring of  $x$  that covers  $M_x$  positions. Since  $\mathbf{u}_M$  may not be unique, the algorithm can return the longest or shortest  $\mathbf{u}_M$ , as required. Thus, if  $C(x, \mathbf{u})$  denotes the number of positions covered by  $\mathbf{u}$ , then  $M_x = C(x, \mathbf{u}_M)$ . Note that  $M$  and  $\mathbf{u}_M$  can also be identified by the  $\mathcal{O}(n \log n)$  all-partial-covers algorithm of Kociumaka *et al.* [59], described in the previous subsection.

The methodology of [62] avoids the use of suffix trees, employing instead only simple sorting and runs. Central to this approach are the  $\mathcal{RSF}$  and  $\mathcal{OLP}$  arrays, introduced in [63] and [62], respectively.

The  $\mathcal{RSF}_x$  (**Repeating Substring Frequency**) array is defined as follows: for  $1 \leq i \leq n$ ,  $\mathcal{RSF}[i]$  is the frequency in  $x[1..n]$  of the repeating substring of length  $\mathcal{LCP}[i]$  that occurs (at least) at the two positions  $\mathcal{SA}[i - 1]$  and  $\mathcal{SA}[i]$ ; that is, the repeating substring

$$x[\mathcal{SA}[i], \dots, \mathcal{SA}[i] + \mathcal{LCP}[i] - 1].$$

Thus, in Figure 1,  $\mathcal{RSF}[4] = \mathcal{RSF}[5] = 3$ , together with  $\mathcal{RSF}[3] < 3$  and  $\mathcal{RSF}[6] < 3$ , tells us that the substring, say  $\mathbf{u}$ , of length  $\mathcal{LCP}[4] = \mathcal{LCP}[5] = 3$  occurs exactly  $\mathcal{RSF}[4] = 3$  times altogether in  $x$  at positions  $\mathcal{SA}[3] = 3, \mathcal{SA}[4] = 6, \mathcal{SA}[5] = 1$ . In [63] a simple algorithm is described that computes  $\mathcal{RSF}_x$  in  $\Theta(n)$  time and space.

Consider a repeating substring  $\mathbf{u} = x[\mathcal{SA}[i].. \mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$  in  $x$  of length  $\ell = \mathcal{LCP}[i] > 1$  that occurs  $k > 1$  times as identified by the maximal sequence  $\mathcal{LCP}[i - 1], \mathcal{LCP}[i], \dots, \mathcal{LCP}[i + k - 1]$ . Then the **Overlapping Positions** array  $\mathcal{OLP} = \mathcal{OLP}_x[1..n]$  specifies, in entry  $\mathcal{OLP}[i]$ , the total number of overlapping positions (overlaps) between **consecutive** occurrences of  $\mathbf{u}$  in  $x$  — where positions  $\mathcal{SA}[i - 1], \mathcal{SA}[i], \dots, \mathcal{SA}[i + k - 1]$  are not in general in ascending sequence and so must somehow be ordered. For example, in Figure 1,  $\mathcal{OLP}[4] = 1$  tells us that the  $\mathcal{RSF}[4] = 3$  occurrences of the substring  $\mathbf{u}$  at positions  $\mathcal{SA}[3..5] = 6, 1, 4$  of length  $\ell = \mathcal{LCP}[4] = \mathcal{LCP}[5] = 3$  — that is, *aba* — have exactly one position ( $i = 6$ ) of overlap in  $x$ .

In [62] two algorithms are described to compute  $\mathcal{OLP}_x$ , one requiring  $\mathcal{O}(n \log n)$  time, which has proved to be incorrect [64], and another requiring  $\mathcal{O}(n^2)$  time. In [64] two additional  $\mathcal{OLP}$  algorithms are described, both requiring  $\mathcal{O}(n^2)$  time in the worst case, but both shown, based on experimental evidence, to execute in linear time in the average case. Apart from  $\mathcal{OLP}$ , all other computations needed to compute the maximal cover require worst-case linear time.

In [65] Golding *et al.* apply an early  $\mathcal{O}(n^2)$  implementation MAXCOVER of the maximal cover algorithm to protein sequences; surprisingly, they find significant compression in certain cases. Further, a version of MAXCOVER, slightly modified to compute nonextendible repeating substrings, is compared to existing software for this purpose, again in the context of protein sequences, and shown to be an order-of-magnitude faster.

### 3.5. Frequency cover

In [63] a similar, slightly less general, approach to the definition of cover was taken, yielding an algorithm with linear requirements for usage of both time and space.

First identify in  $x$  the set  $U = U_x$  of repeating substrings that are not single letters, that occur a maximum number  $M$  of times in  $x$ , and that cannot be extended to left or right. Then the **frequency cover**  $\mathbf{u}$  of  $x$  is defined to be the longest of the entries in  $U$ . Thus, for  $x = abaababaabaab$ ,  $U = \{ab\}$

because  $ab$  occurs  $M = 5$  times in  $x$ , more than any other substring, and so  $u = ab$  — the same substring chosen by the  $\alpha$ -Partial Cover algorithm (Section 3.3) for  $\alpha = 10$ .

From this example, we see that the frequency cover may not in fact be the substring that covers a maximum number of positions ( $aba$  of length 3 is not in  $U$  but covers 11 positions). Furthermore, not all strings have a frequency cover (for example,  $x = abc$  or  $abaca$ ), while some strings have multiple frequency covers, perhaps with different properties ( $x = ababacb$  gives rise to  $u_1 = aba$  covering five positions and  $u_2 = bab$  covering six positions).

In order to compute the frequency cover efficiently, it turns out to be convenient to make use of the  $\mathcal{RSF}$  (Repeating Substring Frequency) array for  $x$ , defined in Section 3.4. For example, in Figure 1  $\mathcal{RSF}[2] = 5$  tells us that the substring of length  $\mathcal{LCP}[2] = 1$  occurring at position  $\mathcal{SA}[2] = 3$  — that is,  $a$  — occurs 5 times in  $x$ . Similarly, since  $\mathcal{RSF}[5] = 3$ , we know that the substring of length  $\mathcal{LCP}[5] = 3$  occurring at position  $\mathcal{SA}[5] = 4$  — that is,  $aba$  — occurs 3 times in  $x$ .

### 3.6. Approximate covers

These algorithms generally depend on counting the minimum number of *edit operations* (insertion, deletion or substitution of a single letter) required to transform one string  $x$  into another string  $x'$ , where these operations may have different weights associated with them. In the following examples, exactly one edit operation transforms  $x \rightarrow x'$  (and of course *vice versa*), implying that the “distance” between  $x$  and  $x'$  is one:

- **insertion**: Insert  $c$  at position 2 of  $x = ab$  to form  $x' = acb$ ;
- **deletion**: Delete  $c$  at position 2 of  $x = acb$  to form  $x' = ab$ ;
- **substitution**: Change  $c$  to  $b$  at position 2 of  $x = ac$  to form  $x' = ab$ .

More generally, given strings  $x$  and  $x'$ , we define *edit distance* ( $E$ )  $E(x, x')$  to be the minimum number of edit operations, weighted edit distance ( $WE$ ) if different weights are assigned to different edit operations, *Levenshtein distance* ( $L$ )  $L(x, x')$  the minimum number of insertions and deletions, and *Hamming distance* ( $H$ )  $H(x, x')$  the minimum number of substitutions — required to transform  $x$  into  $x'$  (and *vice versa*)<sup>3</sup>. We use  $D$  to indicate any one of  $E, L, H$ . For details see [2, Sect. 2.2].

The idea of an approximate cover of a string was apparently introduced by Sim *et al.* [66] (not available in English), then by Zhang and Blanchet-Sadri [67], whom we follow here. Given a string  $x$  of length  $n$  and a set  $U_{t,k} = \{u_1, u_2, \dots, u_t\}$  of  $t$  strings of identical length  $k < n$ ,  $U_{t,k}$  is said to be a *d-approximate k-cover* of  $x$  for some integer  $d \geq 0$  if there exists a set  $V = \{u_1, u_2, \dots, u_r\}$  of  $r$  distinct nonempty strings, not necessarily of equal length, such that

- $V$  is a cover of  $x$ ;
- for every  $u \in U_{t,k}$ , there exists  $v \in V$  such that  $D(u, v) \leq d$ ;
- for every  $v \in V$ , there exists  $u \in U_{t,k}$  such that  $D(u, v) \leq d$ .

<sup>3</sup>In the literature these definitions vary. Of course deletion/insertion at position  $i$  is just a “substitution”, so both unweighted edit distance and Hamming distance merely count two distinct Levenshtein operations as one.

The authors of [67] then described polynomial-time algorithms that, using the results from [51], compute, for each distance measure  $D^4$ , the minimum integer  $d$  such that  $U_{t,k}$  is a set of  $d$ -approximate  $k$ -covers of  $x$ . Unfortunately, this result was incorrect: it was later shown in [52] that to determine whether or not any given set  $V$  was indeed a minimum  $k$ -cover was NP-complete (see Section 3.2). Nevertheless, practical algorithms were proposed to compute approximations of  $V$  ([67], [53]), and, as discussed below, several variants of  $V$  have been proposed.

Although not a cover problem, in 2001 Sim *et al.* [68] introduced a related and more tractable problem. Given strings  $x[1..n]$  and  $u[1..m]$ ,  $m < n$ , for  $D = E, H$  consider partitions  $x = u_1 u_2 \cdots u_r$  such that  $D(u_i, u) \leq t$ ,  $1 \leq i < r$ , and  $D(u_r, u') \leq t$  for some prefix  $u'$  of  $u$ . For each such partition,  $p = |u|$  is said to be a ***t*-approximate period** of  $x$ . A polynomial-time algorithm is described to compute a minimum integer  $t$  for which such a partition exists.

In 2005 Christodoulakis *et al.* [69] studied a related cover problem: given strings  $x[1..n]$  and  $u[1..m]$ ,  $m < n$ , consider arrangements of copies of  $u$  placed so as to overlay all positions in  $x$ . Over all copies of  $u$  in each arrangement, determine the total distance  $D$  resulting from mismatches; then an arrangement that minimizes  $D$  is a ***u*-approximate cover** of  $x$ . For  $D = H/E/WE^5$ , the authors describe algorithms to compute all  $u$ -approximate covers in  $x$  requiring time  $\mathcal{O}(mn)/\mathcal{O}(m(n+m))/\mathcal{O}(mn^2 + n^2)$ , respectively.

More recently, in 2019 Amir *et al.* [70] introduced the ***Approximate Cover Problem (ACP)***: find a “best” approximate cover of a given string  $x$  of length  $n$ ; that is, identify a string  $y$  of length  $m < n$  whose copies cover a string  $z$ , also of length  $n$ , in such a way that, over all choices of  $y$ ,  $H(x, z)$  is minimized. By considering a relaxed version of ACP, they show that ACP itself is NP-hard. They then discuss two relaxations of the ACP problem, where either a partial or a full ordered list of occurrences of the possible cover of  $x$  is given. They show that both these problems have polynomial time solutions.

In [71] the ***Cover Recovery Problem (CRP)*** is introduced: given a string  $x' = x'[1..n]$  that results from the ***approximate*** covering of  $x[1..n]$  by an unknown string  $u$  of known length  $m$ , output a “small size set”  $S$  of strings of length  $m$  such that  $u \in S$ . In [72] further results are presented to assist in understanding the overall complexity of ACP.

Then in 2019, Guth [73], building on previous work [74, 75], introduced a relaxed version of the enhanced cover (Section 3.3). Given a non-negative integer  $k$ , a ***k*-approximate enhanced cover** ( $k$ -AEC)  $y$  of  $x$  is a border of  $x$  such that the total number of positions covered by approximate occurrences of  $y$  in  $x$  exceeds those covered by approximate occurrences of any other border of  $x$ . Computation of all  $k$ -AECs is shown to require  $\mathcal{O}(n^2)$  time. A “relaxed”  $k$ -AEC is also considered, computable in time  $\mathcal{O}(n^3)$ .

In [76, 77], given strings  $x = x[1..n]$  and  $u$ , an integer  $k \in 0..n - 1$ , and distance measure  $D$ , Kedzierski & Radoszewski study the ***(D, k)*-coverage of u in x**; that is, the number of positions of  $x$  that lie within a substring  $v$  such that  $D(u, v) \leq k$ . For given  $k$ , they describe an  $\mathcal{O}(n^2)$ -time algorithm to compute  $(H, k)$ -coverage for all substrings  $u$  of  $x$ ,  $\mathcal{O}(n \log^{1/3} n k^{2/3} \log k)$  for all prefixes. For  $D = E, WE$ , they describe algorithms to compute  $(D, k)$ -coverage for all substrings  $u$

<sup>4</sup>In [67] the authors define Edit Distance as Levenshtein distance defined here and *vice versa*.

<sup>5</sup>In [69] the authors use Edit Distance and Levenshtein distance interchangeably.

in time  $\mathcal{O}(n^3)$  and  $\mathcal{O}(n^3\sqrt{n\log n})$ , respectively. They also show that it is NP-hard to check whether or not a given  $x$  has a  $k$ -approximate cover (or seed) of given length  $\ell$ , even on a binary alphabet.

It is noteworthy that, with the exception of the approach of Guth [73], the methodologies described in this subsection avoid the requirement that the approximate cover should be a border of  $x$  — which as we have seen has an average length of at most 1.64. No doubt the maximal cover of most strings — for example,  $u = aba$  for  $x = acabaababaaac$  — will be unrelated to any border.

### 3.7. 2-Dimensional covers

In 1996 Iliopoulos & Korda [78] described an  $\mathcal{O}(\log \log n)$ -time parallel algorithm on the CRCW PRAM model to determine whether a given  $n \times n$  square matrix  $T$  is superprimitive — that is, whether there exists a square submatrix  $S$  of  $T$  such that every position in  $T$  lies within an occurrence of  $S$ ; in other words, such that  $S$  is a (2D) **cover** of  $T$ . If so, then they return a smallest cover  $S$ . Then in 1998 Crochemore *et al.* [79] showed how to compute all the covers of a given square matrix  $T$  by presenting an  $\mathcal{O}(n^2)$  time algorithm to compute all square submatrices  $P$  of  $T$  that cover  $T$  — a result based on the Aho–Corasick [80] automaton and “gap” monitoring techniques.

More recently the problem has been generalized from square to  $m \times n$  rectangular matrices  $T$ , with  $N \equiv mn$ : in 2019 Popa & Tanasescu [81] describe an average-case  $\mathcal{O}(N)$ -time algorithm to compute a smallest 2D cover by rectangular submatrices  $S$  of a given  $T$ , as well as a worst-case  $\mathcal{O}(N^2)$ -time algorithm to compute all 2D covers of  $T$ . They propose applications such as extraction of textures from images, as well as to image compression and crystallography.

A very recent paper [82] by Charalampopoulos *et al.* considers two forms of cover of a given  $m \times n$  matrix  $T$ : the 2D cover described above and a 1D cover by a vector  $S$  whose occurrences in  $T$  are considered both vertically and horizontally. They present several new results:

- The smallest 2D cover can be computed in time  $\mathcal{O}(N)$ .
- All 2D covers can be computed in time  $\mathcal{O}(N^{4/3})$ .
- All 1D covers can be computed in time  $\mathcal{O}(N)$ .

In [83] Radoszewski *et al.* propose another form of cover called the **tile cover**; that is, a string  $S$  that covers  $T$  by **non-overlapping** instances of  $S$  or its transpose  $S^T$ . The authors consider two forms of tile cover: 2D-string and 1D-string tile cover (this differs from the 1D cover proposed in [82] by disallowing overlaps). They propose an  $\mathcal{O}(N)$ -time algorithm to compute all 1D tile covers of  $T$ , and an  $\mathcal{O}(N^{1+\epsilon})$ ,  $\epsilon > 0$ , algorithm to compute all 2D-tile covers of  $T$ .

### 3.8. Specialized covers

In [84] Alzamel *et al.* introduce the  **$k$ -anticover** of given  $x = x[1..n]$ ; that is, for a given integer  $k \geq 2$ , a set  $S = S_k$  of increasing positions  $i$  in  $x$  identifying substrings  $x[i..i+k-1]$ , constrained to be distinct, such that every position in  $x$  is contained in an entry from  $S$  — in other words, such that  $S$  “covers”  $x$ . For example [84], given  $x = abbbaaaaabab$  of length  $n = 12$  and  $k = 3$ ,  $S_3 = \{1, 3, 5, 8, 10\}$  is a 3-anticover of  $x$ , identifying distinct substrings  $abb, bba, aaa, aab, bab$  that

cover  $x$ ; on the other hand, no string  $x = uvu$  has a  $|u|$ -anticover. It is shown in [84] that for  $k \geq 3$  it is NP-hard to determine whether or not a  $k$ -anticover of  $x$  exists, while a polynomial-time solution exists for  $k = 2$ . In [85] Amir *et al.* introduce three variants of the  $k$ -anticover problem, as follows:

- **MaxkAnticover:** find a set  $S_k$  that *maximizes* the number of covered positions in  $x$ ;
- **MinRepkAnticover:** if there exists no  $S_k$ , then find a set  $S'_k$  of  $k$ -strings that allows duplicates and covers  $x$  with a minimum number of repeats of any one entry;
- **MinAnticover:** find the smallest  $k$  such that there exists a  $k$ -anticover of  $x$ .

All of these variants are also shown to be NP-hard; however, polynomial-time approximation algorithms are described for each.

In [86] Matsuda *et al.* introduce the **Abelian cover**; that is, a  $k$ -cover of  $x$  in which each entry has the same Parikh vector  $P = P_x$ . For example, for  $k = 3$ ,  $x = abaab$  has Abelian cover  $(aba, aab)$ , each with Parikh vector  $P = (2, 1)$ . They describe an  $\mathcal{O}(n)$ -time algorithm to compute the longest Abelian cover, whenever it exists, of given  $x = x[1..n]$ , as well as an  $\mathcal{O}(n^2)$ -time algorithm to compute an  $\mathcal{O}(n^2)$  representation of *all (possibly exponential)* Abelian covers of  $x$ .

Similarly, in [87], Grossi *et al.* introduce the **cyclic cover** of  $x$ ; that is, any substring  $u$  whose rotations cover  $x$ . In the above example  $x = abaab$ , therefore, every rotation of  $aba$  is a cyclic cover of  $x$ . The authors describe an  $\mathcal{O}(n \log n)$  time algorithm to compute all the cyclic covers of a string. A recent paper [88] improves the time requirement for this problem to  $\mathcal{O}(n)$ .

Recall that for  $1 \leq i \leq j \leq n$ ,  $u = x[i..j]$  is a **palindrome** at *centre*  $(i + j)/2$  with **radius**  $(j - i + 1)/2$  if  $x[i + h] = x[j - h]$  for every  $h = 0, 1, \dots, (j - i)/2$  — **maximal** if there exists no palindrome of greater radius at the same centre. For given  $k \in 1..n$ , we say that  $x$  has a **palindromic cover**  $PC_{x,k}$  of **size**  $k$  if every position of  $x$  lies within a palindrome of radius  $k/2$ . Of course every single entry  $x[i]$  is a palindrome of radius  $1/2$ , and so  $x$  always has palindromic cover  $PC_{x,1}$ . In [89] I *et al.* describe a  $\mathcal{O}(n)$ -time and space algorithm to compute the *smallest*  $k$  such that  $PC_{x,k}$  is a palindromic cover of given  $x$ .

In [90], Radoszewski *et al.*, study covers in both directed and undirected labeled trees. They propose an  $\mathcal{O}(n \log n / \log \log n)$ -time algorithm to compute all covers of a directed (rooted) tree, and an  $\mathcal{O}(n^2)$ -time and space algorithm to compute all covers of an undirected labeled tree.

Recently, in [91] Charalampopoulos *et al.* introduce the **subsequence cover (or s-cover)** of  $x$ ; that is, a substring  $u$  whose occurrences as subsequences cover all the positions in  $x$ . They present a linear-time algorithm to test whether a given string  $v$  is an s-cover of a word  $x$ , where  $x$  is defined on polynomially-bounded integer alphabet. They then present a  $\mathcal{O}(n)$ -time algorithm to compute the shortest s-cover of the given  $x$ , where  $x$  is defined on a constant sized alphabet.

### 3.9. Extensions to indeterminate & weighted strings

Substantial work has been done on extending covering algorithms to indeterminate strings.

Of course every cover is a border: we have noted above [21] that the *expected length* of the maximum border of a regular string does not exceed 1.64. For partial words Iliopoulos *et al.* showed

[92] that the *expected number* of borders was less than 3.5; for indeterminate strings Bari *et al.* [93] showed that this quantity was less than 29.1746.

In 2003 Iliopoulos *et al.* [94] described two algorithms to compute the border array of a partial word, both requiring  $\mathcal{O}(n^2)$  time in the worst case,  $\mathcal{O}(n)$  time on average. In the same year [95] Holub & Smyth described border array calculations with the same quadratic time complexity on both partial words and indeterminate strings. Holub & Smyth also make a distinction between **quantum** borders, which allow indeterminate letters to match in more than one way, and **deterministic** borders, in which only a single match is allowed. They give the example  $x = a**c$ , which has two quantum border pairs,  $(a*, *c)$  and  $(a**, **c)$ , requiring  $x[2]$  to match both  $c$  and  $a$ , but only one deterministic border pair — either  $(a*, *c)$ , requiring  $x[2] = c$ , or the pair  $(a**, **c)$ , requiring  $x[2] = a$ .

In 2008 Antoniou *et al.* [96] introduce the idea of a ***q-conservative*** indeterminate string — that is, a string  $x$  containing at most  $q \geq 0$  indeterminate letters. They suggest use of the Aho-Corasick automaton [80] to determine whether, for given nonnegative integers  $q$ ,  $q'$  and  $m$ ,  $q$ -conservative  $x$  has a  $q'$ -conservative cover of length  $m$ . There is no clear description of an algorithm. A subsequent paper [97] discusses covers of DNA strings on  $(a, c, g, t)$ .

In 2009 Bari *et al.* [93] present an *average-case*  $\mathcal{O}(n)$ -time algorithm, also using the Aho-Corasick automaton, to compute all the covers of a regular or indeterminate string  $x$  based on the border array algorithm in [95]. (For indeterminate strings, the cover also may be indeterminate.) They extend their algorithm to compute the cover array in  $\mathcal{O}(n^2)$  time.

More recently, Crochemore *et al.* in [98] consider the problem of finding a shortest **regular** cover of an indeterminate string  $x$ , showing that the computation is NP-complete even over strings  $x$  that are restricted to partial words. However, they also describe “near-optimal” FPT (Fixed Parameter Tractable) algorithms for both partial words and the general case, based on knowledge of a parameter  $k$  — the number of non-regular letters in  $x$ .

Weighted strings introduce a new form of ambiguity, as the following example, taken from Zhang *et al.* [99], demonstrates. In the weighted string

$$x = (a, 50; c, 25; g, 25)g(a, 60; c, 40)(a, 25; c, 25; g, 25; t, 25)c,$$

the pattern  $u = agc$  matches two overlapping substrings:

$$x[1..3] = (a, 50; c, 25; g, 25)g(a, 60; c, 40)$$

and

$$x[3..5] = (a, 60; c, 40)(a, 25; c, 25; g, 25; t, 25)c$$

with probabilities  $p_1 = 0.5 \times 1 \times 0.4 = 0.2$  and  $p_2 = 0.6 \times 0.25 \times 1 = 0.15$ , respectively. In the first of these,  $u[3] = c$  matches with  $x[3]$ , but in the second,  $u[1] = a$  provides the match with  $x[3]$ . If overlapping matches that depend on this ambiguous use of an indeterminate letter are allowed, then the matching is called **loose**; if not, then **strict**. (See above, quantum/deterministic.) In [99] the authors outline  $\mathcal{O}(n^2)$ -time algorithms to compute all the covers of weighted strings, using both loose and strict matching.

An  $\mathcal{O}(n)$ -time algorithm to compute the covers of a weighted string is also given in [100], based on prior calculation of a **weighted suffix tree** in time  $\mathcal{O}(\sigma n)$ . A recent paper [101] describes a more efficient cover calculation based on a “weighted index”.

## 4. Seeds of strings

As defined earlier, the *seed* of a string  $x$  is a proper substring  $u$  of  $x$  that is a cover of a superstring  $w$  of  $x$ . The notion of seed was first introduced in [6, 7] by Iliopoulos *et al.*, where they describe an  $\mathcal{O}(n \log n)$ -time algorithm to compute all the seeds of a given string  $x[1..n]$ , by computing a linear representation of seeds. Berkman *et al.* in [102] present a parallel algorithm to compute all seeds in  $\mathcal{O}(\log n)$  time and  $\mathcal{O}(n^{1+\epsilon})$  space, using  $n$  processors in the CRCW PRAM model. Then in [19] Christou *et al.* present an alternate sequential  $\mathcal{O}(n \log n)$  algorithm for computing the shortest seed of a given string. In [103] Smyth poses the question whether all seeds for the given string can be computed in time linear in its length. In 2012 Kociumaka *et al.* [104] answer this question in the affirmative by presenting the first linear time algorithm to compute seeds — though based on the assumption of an integer alphabet. Their algorithm was complex and required constructing a representation of seeds on two suffix trees. In 2020, the same authors [8] present a solution to the same problem that uses a much simpler approach called the *package representation* — again based on an integer alphabet. The authors define a *package* to be a collection of consecutive prefixes of a substring of  $w$ ; that is, a package is defined as

$$\mathit{pack}(i, j_1, j_2) = \{w[i..j] : j_1 \leq j \leq j_2\},$$

where  $i \leq j_1 \leq j_2 \leq |w|$ . If  $\mathcal{L}$  is a set of ordered integer triples, they then define

$$\mathit{PACK}(\mathcal{L}) = \bigcup_{(i, j_1, j_2) \in \mathcal{L}} \mathit{pack}(i, j_1, j_2)$$

Their solution outputs the set  $\mathcal{L}$  such that the seeds of  $w$  are exactly the elements of  $\mathit{PACK}(\mathcal{L})$ . Furthermore, all packages in the representation are pairwise disjoint; that is, each seed belongs to exactly one package. It turns out that packages correspond to paths in the suffix trie that can be easily stored using the suffix tree. This fact and the connection between seeds and subword complexity both contribute to the reduced linear time complexity of the seed computation algorithm.

Many other problems on seeds similar to those on covers have also been studied and are outlined below.

### 4.1. Left and right seeds

A *left (right) seed*  $u$  of a string  $x$  is a prefix (suffix) of  $x$  that is also a seed of  $x$ . A *minimal (maximal) left seed array* of  $x[1..n]$  is an integer array of length  $n$  whose  $i$ -th element is the minimum (maximum) length of the left seed of  $x[1..i]$ . The *minimal (maximal) right seed array* is defined analogously. See Figure 2 for an example presented in [105]. In [19, 106] Christou *et al.* present a linear-time algorithm to compute the minimum and maximum left seed arrays of  $x$ . Both these algorithms rely on the linear time computation of the period and cover arrays of  $x$  (see introduction to Section 3) in order to achieve  $\mathcal{O}(n)$  running times. In the same paper, they also give an  $\mathcal{O}(n^2)$  algorithm to compute all the seeds of  $x$  of length at least  $k$  using the precomputed suffix, LCP, period and suffix period arrays of  $x$ . In addition, they present an alternate  $\mathcal{O}(n \log n)$ -time approach to computing the shortest seeds of  $x$  that is based on independent processing of disjoint chains in the suffix tree of  $x$ . Further, by checking



whether the shortest seed has length at least  $m$ , they extend this algorithm to compute the shortest seeds of length at least  $m$  in  $\mathcal{O}(n \log(n/m))$  time. Thus, for sufficiently large  $m = \Theta(n)$ , the running time of the algorithm reduces to  $\mathcal{O}(n)$ .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b
$LS_{min}$	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3
$LS_{max}$	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14
$RS_{min}$	1	2	2	3	3	3	5	3	5	5	3	8	5	3	8
$RS_{max}$	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 2:  $LS_{min}$ ,  $LS_{max}$ ,  $RS_{min}$  and  $RS_{max}$  are the minimal left seed, maximal left seed, minimal right seed and maximal right seed arrays, respectively, computed for the string  $x = abaababaabaabab$  — adapted from [105].

In [107] Christou *et al.* describe an  $\mathcal{O}(n \log n)$ -time algorithm to compute the minimal right seed array. Their solution uses a variant of the partitioning algorithm introduced by Crochemore in [108], as employed by Iliopoulos, Moore & Park in [6], to find the sets of ending positions of all occurrences of each distinct substring in  $x$ . Using this methodology, [107] finds a suffix of each prefix of the string that is covered by some substring, then checks for occurrences of right seeds to compute the minimal right seed array. They also present a simple  $\mathcal{O}(n)$  algorithm to compute the maximal right seed array by detecting border-free prefixes of  $x$ . In addition to these results, the extended journal version [105] of [107] describes algorithms to compute all the left and right seeds of  $x$ . To compute all the left seeds of  $x$  their linear time algorithm uses the maximal cover array and the period array of  $x$ . Since the right seeds of  $x$  are just the left seeds of the reverse string  $x^R$ , all right seeds of  $x$  can be computed in linear time by applying the left seeds algorithm to  $x^R$ .

In 2013 Flouri *et al.* [61] introduce the **enhanced left seed**; that is, a proper prefix  $u$  of  $x$  that occurs at least twice in  $x$  and such that the number of letters in  $x$  which lie within some occurrence of  $u$  in a superstring of  $x$  is a maximum over all such prefixes of  $x$ . Making use of new data structures introduced in the paper, the authors describe an  $\mathcal{O}(n \log n)$ -time algorithm to compute the minimal (shortest length) enhanced left seed of  $x$ . The running time of the algorithm is dominated by the time required to compute some of these data structures. Then they go on to define the **enhanced left-seed array** — an integer array of length  $n$  whose  $i$ -th element is equal to the length of the enhanced left seed of the prefix of length  $i$ . To compute the minimal enhanced left-seed array, they apply the minimal enhanced left seed algorithm repeatedly, thus computing the minimal enhanced left seed of every prefix of  $x$  in  $\mathcal{O}(n^2)$  time.

## 4.2. $\lambda$ -seeds

In [109] Guo *et al.* attempted to extend the  $\lambda$ -covers problem of Section 3.2 to  $\lambda$ -seeds; that is, given  $x[1..n]$  and an integer  $\lambda$ , find all the sets  $U = \{u_1, u_2, \dots, u_\lambda\}$  of substrings of  $x$  such that:

- (1)  $|u_1| = |u_2| = \dots = |u_\lambda|$ ;
- (2) there exists a superstring  $y = vxw$  of  $x$  with  $|v|, |w| < |u_i|$  such that  $y$  can be constructed by concatenating or overlapping elements of  $U$ .

Of course the results presented in [109] are subject to the same difficulties raised by the counterexample of Czajka & Radoszewski [36] (Section 3.2).

## 4.3. Approximate seeds

Here again we make use of the distance measures defined in Section 3.6. In [110], Christodoulakis *et al.* study the approximate seeds of strings under the distance rules  $D = H$ ,  $D = E$  and  $D = WE$  (weighted edit).

They define a string  $s$  to be a *t*-approximate seed of  $x$ ,  $t \in \mathbb{N}$ , if there exist nonempty strings  $s_1, s_2, \dots, s_r$  such that (i)  $D(s, s_i) \leq t$ , for  $1 \leq i \leq r$ , and (ii) there exists a superstring  $y = uxv$  of  $x$ ,  $|u| < |s|$  and  $|v| < |s|$ , that can be constructed by overlapping or concatenated copies of  $s_1, s_2, \dots, s_r$ . They then solve the following three problems:

- (1) *Smallest distance approximate seed problem*: here strings  $x[1..n]$ ,  $s[1..m]$  and a distance function  $D$  are assumed to be given and the minimum  $t$  value is computed. Their solution first computes the distance between  $s$  and every substring  $u$  of  $x$  and then uses a dynamic programming approach to compute  $t_i$  such that  $s$  is a  $t_i$ -approximate seed of  $x[1..i]$ . It follows that  $t_n$  is the minimum  $t$  such that  $s$  is a  $t$ -approximate seed of  $x$ . The algorithm runs in  $\mathcal{O}(mn)$  time for both  $D = H, E$ . However for  $D = WE$ , it requires  $\mathcal{O}(mn^2)$ -time.
- (2) *Restricted smallest approximate seed problem*: In this case the string  $s$  is not given, and so any substring of  $x$  is a candidate for the  $t$ -approximate seed. Their solution to this problem is similar to that used to solve problem (1), but requires significantly more time —  $\mathcal{O}(n^4)$  time for  $D = WE$  and  $\mathcal{O}(n^3)$  time for  $D = H$ .
- (3) *Smallest approximate seed problem*: This problem is a generalization of (2) in that not only is  $s$  not given, it is moreover not required to be a substring of  $x$ . The authors show that this problem is NP-complete for any distance rule  $D$  by reduction from the NP-complete shortest common supersequence (SCS) problem [111, 112].

Further finite automaton-based algorithms for problems (1) and (2), under Hamming distance ( $D = H$ ) bounded by  $k$ , were described in several contributions by Guth *et al.* [113, 114, 115]. An algorithm for problem (3) was included in Guth's doctoral dissertation [116], which also included experimental evaluation of these algorithms.

In [76], Kedzierski and Radoszewski propose efficient algorithms for computing (many) variants of approximate covers and seeds and improve upon the complexities of previous algorithms. They

show that their solutions are particularly efficient if the number (or total cost) of the allowed errors is bounded. In the context of seeds, they notably present  $\mathcal{O}(n^2k)$  and  $\mathcal{O}(n^3\sqrt{n}\log n)$  time algorithms to solve the above problem (2) for  $D = H$  and  $D = WE$ , respectively. They also show that for  $D = H$ , problem (3) remains NP-hard even when the length of  $s$  is fixed, a result that holds even for strings on a binary alphabet.

#### 4.4. Partial seeds

In this section we discuss the notion of partial seeds introduced in [117] by the same authors (Kociumaka *et al.*) who introduced the partial covers [59] discussed in Section 3.3.

Let  $C(\mathbf{u}, \mathbf{x})$  denote the number of positions covered by (full) occurrences of  $\mathbf{u}$  in  $\mathbf{x}$ . Then the non-empty prefix (suffix) of  $\mathbf{x}$  that is also the suffix (prefix) of  $\mathbf{u}$  is called the **left (right) overhanging** occurrence of  $\mathbf{u}$  in  $\mathbf{x}$ .  $S(\mathbf{u}, \mathbf{x})$  is the number of positions covered by the full, left and right overhanging occurrences of  $\mathbf{u}$  in  $\mathbf{x}$ . Then  $\mathbf{u}$  is an  $\alpha$ -**partial seed** of  $\mathbf{x}$ , if  $S(\mathbf{u}, \mathbf{x}) \geq \alpha$ . For example, if  $\mathbf{x} = \text{abaababaaaaba}$ , then  $S(\text{aba}, \mathbf{x}) = 11$  and  $S(\text{abaa}, \mathbf{x}) = 11$ . Therefore both are 11-partial seeds of  $\mathbf{x}$ , but  $\text{aba}$  is the shortest one. The authors present an  $\mathcal{O}(n \log n)$  algorithm to compute all shortest  $\alpha$ -partial seeds of  $\mathbf{x}$ . Their solution uses the augmented suffix tree (cover suffix tree), originally introduced in [59], to compute  $\alpha$ -partial covers, with some additional nodes. They also describe an  $\mathcal{O}(n \log n)$  time algorithm to compute a factor  $\mathbf{u}$  of  $\mathbf{x}$ , given an interval  $[\ell, r]$ ,  $0 \leq \ell \leq r \leq n$ , such that  $|\mathbf{u}| \in [\ell, r]$  and which maximizes  $S(\mathbf{u}, \mathbf{x})$ . Recently, by giving a linear time construction of the cover suffix tree data structure, Radoszewski in [60] shows that all  $\alpha$ -partial seeds can be computed in  $\mathcal{O}(n)$ -time.

#### 4.5. Extensions to indeterminate and weighted strings

In Section 3.9 we discussed  $q$ -conservative indeterminate strings, and the problem of finding covers in such strings. Here we mention the  $\lambda$ -conservative seeds problem introduced by Antoniou *et al.* in [96]: given a  $q$ -conservative indeterminate string  $\mathbf{x}$  and  $\lambda \in \mathbb{Z}^+$ , a  $\lambda$ -conservative seed is a seed of  $\mathbf{x}$  of length  $\lambda$ . Making use of the Aho-Corasick automaton [80], the authors describe an  $\mathcal{O}(n\lambda)$ -time algorithm to compute the  $\lambda$ -conservative seeds (if they exist) of  $\mathbf{x}$ .

Section 3.9 also discussed weighted strings and cover algorithms for them. In [99] Zhang *et al.* discuss *loose* and *strict* matching (see Section 3.9) and present two cover algorithms for weighted strings based on these string matching variations. In the same paper, they also propose  $\mathcal{O}(n^2)$ -time algorithms that compute all seeds in the given weighted string based on these matchings.

## 5. Open problems

As suggested in the Introduction, a central motivation for the study of covers and seeds is the ubiquitous requirement to find compressed representations of long strings that moreover disclose patterns — some sort of “meaning” — not evident in their original linear formulation. In this context we present here a collection of open problems arising from the work surveyed above, some suggested by the authors themselves, some of them new.

## 5.1. Covers

**Find String on Minimum Alphabet (3):** [42] describes a linear-time approach to determining a string corresponding to a given border array or prefix array, but not necessarily on a minimum alphabet. Can this improvement be achieved, also in linear time?

**Shortest Covers, All Rotations (3.1):** (1) The question of the space required for shortest cover computation has also been raised. In [118] two space-efficient near-linear randomized algorithms are described that with high accuracy compute the shortest cover of given  $x$ . Can the shortest cover be computed in  $\text{polylog}(n)$  space?

(2) [43, 44, 46] all employ  $\mathcal{ST}_{x^3}$  to represent the seeds of  $x$ . Can a more direct approach be found, possibly replacing the suffix tree, possibly reducing processing time?

**$k$ -Covers (3.2):** In view of the NP-completeness of the original  $k$ -cover problem, and the difficulty discovered in [36] with its replacement, further results in this challenging area would be very welcome.

**$\lambda$ -Covers (3.2):** (1) The complex work of Radoszewski & Straszyński [58] deals efficiently with the case  $\lambda = 2$ . Their solution also deals with the generalized case of  $\lambda > 2$ ; however, for each unit increase in  $\lambda$ , the running time complexity of the solution increases by a factor of  $n$ . Does a solution with better running time for  $\lambda > 2$  exist?

(2) In [58] the authors propose the following problems:

- (a) Can a shortest 2-cover be computed in linear time?
- (b) Can we efficiently compute a variation of the 2-covers (and  $\lambda$ -covers) problem, in which the factors that cover the string are of different lengths?

**Maximal Covers (3.4):** (1) As noted in Section 3.3, the all-partial-covers can be computed in  $\mathcal{O}(n)$ -time using the linear time construction of the cover suffix tree data structure. Can we compute the all-partial-covers/maximal covers without the need to use annotated suffix trees?

(2) Are the maximal covers of practical use for the compact representation of any classes of string? If so, can the computation of this representation be iterated?

**ACP Problems (3.6):** In [70, 71] the authors propose related problems:

- (1) Does ACP remain NP-hard on a constant alphabet?
- (2) What is the effect on ACP complexity if distance metrics other than Hamming are used?
- (3) In addition to RACP, are there other relaxations of ACP solvable in polynomial time?

**Specialized Covers (3.8):** In [86] the authors ask whether all the Abelian covers of given  $x$  can be computed in less than  $\mathcal{O}(n^2)$  time. The question has to some degree been answered affirmatively by Kociumaka *et al.* [119], who describe an  $\mathcal{O}(n^2/\log n)$  algorithm for this and other related problems, given a constant-sized alphabet. However, in view of the recent proof [120] that the closely related Abelian squares problem is “3SUM” hard, it seems unlikely that a clearly more efficient algorithm, free of the  $n^2$  factor, can be found. Also, the following questions arise:

- (1) How often can strings on a small alphabet be covered by an Abelian cover?
- (2) Can these “coverable” strings be characterized in a useful way?

Similarly, how many strings possess a cyclic cover — significantly more than those with just a cover?

## 5.2. Seeds

**Left and Right Seeds (4.1):** In [19, 107] the authors pose the following problem: given an integer array  $A$  of length  $n$ , determine whether or not  $A$  is the minimal left-seed (resp. right-seed) array of some string and, if so, construct one such string. In addition, the following questions are of interest:

- (1) In the above problem, what would be the minimum number of distinct letters required to build such a string? Can such strings always be constructed over a bounded alphabet?
- (2) Can we compute the minimal right seed array in linear time?

**Enhanced Left Seeds and Enhanced Left Seeds Array (4.1):** As noted in [61], the same problem arises here as with the Abelian covers calculation: can the minimal enhanced left-seed array be computed in time clearly less than order  $n^2$ ? Moreover:

- (1) Does an  $\mathcal{O}(n \log n)$  or  $\mathcal{O}(n)$  algorithm exist to compute this array?
- (2) What is the complexity of an optimal algorithm to compute the *maximal* enhanced left-seed array? Does an  $\mathcal{O}(n \log n)$  or  $\mathcal{O}(n)$  algorithm exist for this problem?

**$\lambda$ -Seeds (4.2):** As with  $\lambda$ -covers, can the  $\lambda$ -seeds problem be usefully reformulated?

**Approximate Seeds (4.3):** Do finite automata-based solutions exist for problems (1) and (2) under  $D = E, WE$ , and bounded by a constant  $k$ ? If so what are the time complexities of these solutions?

**Partial Seeds (4.4):** Can we compute partial seeds using the  $\mathcal{SA}$  and  $\mathcal{LCP}$  arrays? If so, can they also be computed in  $\mathcal{O}(n)$  time?

## 6. Conclusion

In this paper we have attempted to bring together in an organized fashion all the results related to covers/seeds published since the invention of these concepts more than 30 years ago. That they have been so much studied testifies to their current relevance as well as to their potential future impact on the development of combinatorics on words and string algorithms. We anticipate that further study of the numerous open problems stated here will lead to significant advances in these fundamental computational areas.

## Acknowledgements

**Revisions:** The authors wish to acknowledge the fine work of several referees whose insightful and knowledgeable commentary has resulted in significant improvement to this paper.

**Funding:** The second author was funded by the Natural Sciences & Engineering Research Council of Canada [Grant Number 10536797].

**Conflict of Interest:** Both authors declare that they have no conflict of interest.

**Informed Consent:** This article does not contain any studies on human participants or animals performed by any of the authors.

## References

- [1] Thue A. Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, 1906. 7:1–22.
- [2] Smyth B. *Computing Patterns in Strings*. Pearson/Addison–Wesley, 2003. ISBN 9780201398397.
- [3] Crochemore M, Hancart C, Lecroq T. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/CBO9780511546853.
- [4] Watson JD, Crick FHC. Molecular structure of nucleic acids. *Nature*, 1953. 171:737–738. URL <https://doi.org/10.1038/171737a0>.
- [5] Apostolico A, Ehrenfeucht A. Efficient Detection of Quasi-periodicities in Strings. Technical Report 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990.
- [6] Iliopoulos CS, Moore DW, Park K. Covering a String. In: Proc. 4th CM–SIAM Symp. on Discrete Algorithms (SODA), volume LNCS 684. 1993 pp. 54–62. URL <https://doi.org/10.1007/BFb0029796>.
- [7] Iliopoulos CS, Moore DW, Park K. Covering a String. *Algorithmica*, 1996. 16(1):288–297. doi: <https://doi.org/10.1007/BF01955677>.
- [8] Kociumaka T, Kubica M, Radoszewski J, Rytter W, Waleń T. A Linear-Time Algorithm for Seeds Computation. *ACM Transactions on Algorithms*, 2020. 16(2):27/1–27/23. doi: <https://doi.org/10.1145/3386369>.
- [9] Manber U, Myers G. Suffix Arrays: A New Method for On–Line String Searches. In: Proc. 1st CM–SIAM Symp. on Discrete Algorithms (SODA). 1990 pp. 319–327.
- [10] Manber U, Myers G. Suffix Arrays: A New Method for On–Line String Searches. *SIAM J. Computing*, 1993. 22:935–948. URL <https://doi.org/10.1137/0222058>.
- [11] Kärkkäinen J, Sanders P. Simple linear work suffix array construction. In: Proc. 30th International Colloquium on Automata, Languages, and Programming, volume LNCS 2719. 2003 pp. 943–955. doi: [https://doi.org/10.1007/3-540-45061-0\\_73](https://doi.org/10.1007/3-540-45061-0_73).
- [12] Kärkkäinen J, Sanders P, Burkhardt S. Linear work suffix array construction. *J. ACM*, 2006. 53(6):918–936. URL <https://doi.org/10.1145/1217856.1217858>.
- [13] Kasai T, Lee G, Arimura H, Arikawa S, Park K. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proc. 12th Annual Symp. Combinatorial Pattern Matching (CPM). 2001 pp. 181–192. doi: [https://doi.org/10.1007/3-540-48194-X\\_17](https://doi.org/10.1007/3-540-48194-X_17).

- [14] Smyth WF. Computing regularities in strings: A survey. *European J. Combinatorics*, 2013. **34**(1):3–14. doi:<https://doi.org/10.1016/j.ejc.2012.07.010>.
- [15] Abouelhoda MI, Kurtz S, Ohlebusch E. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2004. **2**(1):53–86. doi:[https://doi.org/10.1016/S1570-8667\(03\)00065-0](https://doi.org/10.1016/S1570-8667(03)00065-0).
- [16] Weiner P. Linear Pattern Matching Algorithms. In: Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT). 1973 pp. 1–11. doi:10.1109/SWAT.1973.13.
- [17] Apostolico A. The myriad Virtues of Subword Trees. In: Combinatorial Algorithms on Words, NATO ISI Series. Springer–Verlag, 1985 pp. 85–96. doi:10.1007/978-3-642-82456-2\\_6.
- [18] Puglisi SJ, Smyth WF, Turpin AH. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 2007. **39**(2). URL <https://doi.org/10.1145/1242471.1242472>.
- [19] Christou M, Crochemore M, Iliopoulos CS, Kubica M, Pissis SP, Radoszewski J, Rytter W, Szreder B, Walen T. Efficient seed computation revisited. *Theoret. Comput. Sci.*, 2013. **483**:171–181. doi:<https://doi.org/10.1016/j.tcs.2011.12.078>. Special Issue Combinatorial Pattern Matching 2011.
- [20] Aho AV, Hopcroft JE, Ullman JD. The Design and Analysis of Computer Algorithms. Addison–Wesley, Reading, MA, 1974.
- [21] Alatabbi A, Islam ASMS, Rahman MS, Simpson J, Smyth WF. Enhanced Covers of Regular & Indeterminate Strings using Prefix Tables. *J. Automata, Languages & Combinatorics*, 2016. **21**(3):131–147. URL <https://doi.org/10.25596/jalc-2016-131>.
- [22] Bland W, Kucherov G, Smyth WF. Prefix table construction and conversion. In: Proc. 24th Internat. Workshop on Combinatorial Algs. (IWOCAL), volume LNCS 8288. 2013 pp. 41–53. doi:[https://doi.org/10.1007/978-3-642-45278-9\\\_5](https://doi.org/10.1007/978-3-642-45278-9\_5).
- [23] Smyth WF, Wang S. New perspectives on the prefix array. In: Proc. 15th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 5280. 2008 pp. 133–143. doi:[https://doi.org/10.1007/978-3-540-89097-3\\\_14](https://doi.org/10.1007/978-3-540-89097-3\_14).
- [24] Iliopoulos CS, Radoszewski J. Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties. In: Proc. 27th Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 54. 2016 pp. 8.1–8.12. doi:10.4230/LIPIcs.CPM.2016.8.
- [25] Mhaskar N, Smyth WF. Simple KMP Pattern-Matching on Indeterminate Strings. In: Proc. Prague Stringology Conference (PSC). 2020 pp. 125–133. URL <http://www.stringology.org/event/2020/p11.html>.
- [26] Dehghani H, Lecroq T, Mhaskar N, Smyth WF. Practical KMP/BM style pattern-matching on indeterminate strings. *submitted for publication*, 2022.
- [27] Knuth DE, Morris JH, Pratt VR. Fast pattern matching in strings. *SIAM J. Computing*, 1977. **6**(2):323–350. doi:<https://doi.org/10.1137/0206024>.
- [28] Boyer RS, Moore JS. A fast string searching algorithm. *Communications of the ACM*, 1977. **20**(10):762–772. doi:<https://doi.org/10.1145/359842.359859>.
- [29] Apostolico A, Ehrenfeucht A. Efficient Detection of Quasiperiodicities in Strings. *Theoret. Comput. Sci.*, 1993. **119**(2):247–265. doi:[https://doi.org/10.1016/0304-3975\(93\)90159-Q](https://doi.org/10.1016/0304-3975(93)90159-Q).
- [30] Iliopoulos C, Mouchard L. An  $O(n \log n)$  Algorithm for Computing all Maximal Quasiperiodicities in Strings. In: Proceeding of the Computing: Australasian Theory Symposium (CATS). 1999 pp. 262–272. URL <https://hal.science/hal-00465077>.

- [31] Brodal GS, Pedersen CNS. Finding Maximal Quasiperiodicities in Strings. In: Proc. 11th Annual Symp. Combinatorial Pattern Matching (CPM), volume LNCS 1848. 2000 pp. 397–411. doi:[https://doi.org/10.1007/3-540-45123-4\\_33](https://doi.org/10.1007/3-540-45123-4_33).
- [32] Apostolico A, Farach M, Iliopoulos CS. Optimal superprimitivity testing for strings. *Information Processing Letters*, 1991. **39**(1):17–20. doi:[https://doi.org/10.1016/0020-0190\(91\)90056-N](https://doi.org/10.1016/0020-0190(91)90056-N).
- [33] Breslauer D. An On-Line String Superprimitivity Test. *Information Processing Letters*, 1992. **44**(6):345–347. doi:[https://doi.org/10.1016/0020-0190\(92\)90111-8](https://doi.org/10.1016/0020-0190(92)90111-8).
- [34] Moore D, Smyth WF. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 1994. **50**:239–246. doi:[https://doi.org/10.1016/0020-0190\(94\)00045-X](https://doi.org/10.1016/0020-0190(94)00045-X).
- [35] Moore D, Smyth WF. Correction to: An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 1995. **54**:101–103. doi:[https://doi.org/10.1016/0020-0190\(94\)00235-Q](https://doi.org/10.1016/0020-0190(94)00235-Q).
- [36] Czajka P, Radoszewski J. Experimental Evaluation of Algorithms for Computing Quasiperiods. *Theoretical Computer Science*, 2021. **854**:17–29. doi:<https://doi.org/10.1016/j.tcs.2020.11.033>.
- [37] Iliopoulos CS, Park K. A Work-Time Optimal Algorithm for Computing All String Covers. *Theoretical Computer Science*, 1996. **164**(1&2):299–310. doi:[https://doi.org/10.1016/0304-3975\(96\)00047-3](https://doi.org/10.1016/0304-3975(96)00047-3).
- [38] Li Y, Smyth WF. Computing the Cover Array in Linear Time. *Algorithmica*, 2002. **32**(1):95–106. doi:<https://doi.org/10.1007/s00453-001-0062-2>.
- [39] Crochemore M, Iliopoulos CS, Pissis SP, Tischler G. Cover array string reconstruction. In: Proc. 21st Annual Symp. Combinatorial Pattern Matching (CPM), volume LNCS 6129. 2010 pp. 251–259. doi:[https://doi.org/10.1007/978-3-642-13509-5\\_23](https://doi.org/10.1007/978-3-642-13509-5_23).
- [40] Moosa TM, Nazeen S, Rahman MS, Reaz R. Linear Time Inference of Strings from Cover Arrays using a Binary Alphabet. In: Proc. 6th International Workshop on Algorithms & Computation (WALCOM), volume LNCS 7151. 2013 pp. 1–16. doi:[https://doi.org/10.1007/978-3-642-28076-4\\_17](https://doi.org/10.1007/978-3-642-28076-4_17).
- [41] Moosa TM, Nazeen S, Rahman MS, Reaz R. Inferring Strings From Cover Arrays. *Discrete Mathematics, Algorithms and Applications*, 2013. **05**(02):1360005. doi:10.1142/S1793830913600057.
- [42] Gawrychowski P, Kociumaka T, Radoszewski J, Rytter W, Waleń T. Universal Reconstruction of a String. *Theoret. Comput. Sci.*, 2020. **812**:174–186. doi:<https://doi.org/10.1016/j.tcs.2019.10.027>.
- [43] Crochemore M, Iliopoulos CS, Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. Shortest Covers of all Cyclic Shifts of a String. In: Proc. 14th International Workshop on Algorithms & Computation (WALCOM), volume LNCS 12049. 2020 pp. 69–80. doi:[https://doi.org/10.1007/978-3-030-39881-1\\_7](https://doi.org/10.1007/978-3-030-39881-1_7).
- [44] Crochemore M, Iliopoulos CS, Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. Shortest Covers of all Cyclic Shifts of a String. *Theoret. Comput. Sci.*, 2021. **866**:70–81. doi:<https://doi.org/10.1016/j.tcs.2021.03.011>.
- [45] Crochemore M, Iliopoulos CS, Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. Linear-Time Computation of Shortest Covers of All Rotations of a String. In: Proc. 33rd Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 223. 2022 pp. 22:1–22:15. doi:10.4230/LIPIcs.CPM.2022.22.
- [46] Crochemore M, Iliopoulos CS, Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. Internal Quasiperiod Queries. In: Proc. 27th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 12303. 2020 pp. 60–75. doi:[https://doi.org/10.1007/978-3-030-59212-7\\_5](https://doi.org/10.1007/978-3-030-59212-7_5).



- [47] Matsuoka Y, Aoki T, Inenaga S, Bannai H, Takeda M. Generalized Pattern Matching and Periodicity under Substring Consistent Equivalence Relations. *Theoretical Computer Science*, 2016. **656**:225–233. doi:<https://doi.org/10.1016/j.tcs.2016.02.017>.
- [48] Fine NJ, Wilf HS. Uniqueness Theorems for Periodic Functions. *Proc. American Mathematical Society*, 1965. **16**(1):109–114. doi:<https://doi.org/10.2307/2034009>.
- [49] Kikuchi N, Hendrian D, Yoshinaka R, Shinohara A. Computing Covers under Substring Consistent Equivalence Relations. In: Proc. 27th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 12303. 2020 pp. 131–146. doi:[https://doi.org/10.1007/978-3-030-59212-7\\_10](https://doi.org/10.1007/978-3-030-59212-7_10).
- [50] Amir A, Iliopoulos CS, Radoszewski J. Two strings at Hamming distance 1 cannot be both quasiperiodic. *Information Processing Letters*, 2017. **128**:54–57. doi:<https://doi.org/10.1016/j.ipl.2017.08.005>.
- [51] Iliopoulos CS, Smyth WF. On-line algorithms for  $k$ -covering. In: Proc. 9th Australasian Workshop on Combinatorial Algs. (AWOCA). 1998 pp. 97–106.
- [52] Cole R, Iliopoulos CS, Mohamed M, Smyth WF, Yang L. The complexity of the minimum  $k$ -cover problem. *J. Automata, Languages & Combinatorics*, 2005. **10-5/6**:641–653. doi:10.25596/jalc-2005-641.
- [53] Iliopoulos CS, Mohamed M, Smyth WF. New complexity results for the  $k$ -covers problem. *Information Sciences*, 2011. **181**:2571–2575. doi:<https://doi.org/10.1016/j.ins.2011.02.009>.
- [54] Guo Q, Zhang H, Iliopoulos CS. Computing the Minimum Approximate  $\lambda$ -Cover of a String. In: Proc. 13th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 4209. 2006 pp. 49–60. doi:[https://doi.org/10.1007/11880561\\_5](https://doi.org/10.1007/11880561_5).
- [55] Guo Q, Zhang H, Iliopoulos C. Computing the  $\lambda$ -covers of a string. *Information Sciences*, 2007. **177**(19):3957–3967. doi:<https://doi.org/10.1016/j.ins.2007.02.020>.
- [56] Zhang H, Guo Q, Iliopoulos CS. Algorithms for Computing the  $\lambda$ -Regularities in Strings. *Fundamenta Informaticae*, 2008. **84**:33–49.
- [57] Iliopoulos CS, Perdikuri K, Zhang H. Computing the regularities in biological weighted sequence. *String Algorithmics, NATO Book series, King's College Publications*, 2004. pp. 109–128.
- [58] Radoszewski J, Straszynski J. Efficient Computation of 2-Covers of a String. In: Proc. 28th Annual European Symposium on Algorithms (ESA), volume 173. 2020 pp. 77:1–77:17. doi:10.4230/LIPIcs.ESA.2020.77.
- [59] Kociumaka T, Radoszewski J, Rytter W, Pissis SP, Waleń T. Fast Algorithm for Partial Covers in Words. *Algorithmica*, 2015. **73**(1):217 – 233. doi:<https://doi.org/10.1007/s00453-014-9915-3>.
- [60] Radoszewski J. Linear Time Construction of Cover Suffix Tree and Applications. In: Proc. 31st Annual European Symposium on Algorithms (ESA), volume 274. 2023 pp. 89:1–89:17. doi:10.4230/LIPIcs.ESA.2023.89.
- [61] Flouri T, Iliopoulos CS, Kociumaka T, Pissis SP, Puglisi SJ, Smyth WF, Tyczynski W. Enhanced string covering. *Theoretical Computer Science*, 2013. **506**:102 – 114. doi:10.1016/j.tcs.2013.08.013.
- [62] Mhaskar N, Smyth WF. String Covering with Optimal Covers. *Journal of Discrete Algorithms*, 2018. **51**:26–38. doi:<https://doi.org/10.1016/j.jda.2018.09.003>.
- [63] Mhaskar N, Smyth WF. Frequency covers for strings. *Fundamenta Informaticae*, 2018. **163**(3):275–289. doi:10.3233/FI-2018-1744.

- [64] Koponen H, Mhaskar N, Smyth WF. Improved Practical Algorithms to Compute Maximal Covers. In: (submitted). 2023 .
- [65] Golding GB, Koponen H, Mhaskar N, Smyth WF. Computing Maximal Covers for Protein Sequences. *Journal of Computational Biology*, 2023. **30**(2):149–160. doi:10.1089/cmb.2021.0520.
- [66] Sim JS, Park K, Kim SR, Lee JS. Finding Approximate Covers of Strings. *Journal of KIISE: Computer Systems and Theory*, 2002. **29**(1):16–21.
- [67] Zhang L, Blanchet-Sadri F. Algorithms for Approximate  $k$ -Covering of Strings. *Int. J. Found. Comput. Sci.*, 2005. **16**(6):1231–1251. URL <https://doi.org/10.1142/S0129054105003789>.
- [68] Sim JS, Iliopoulos CS, Park K, Smyth WF. Approximate period of strings. *Theoret. Comput. Sci.*, 2001. **262**:557–568. doi:[https://doi.org/10.1016/S0304-3975\(00\)00365-0](https://doi.org/10.1016/S0304-3975(00)00365-0).
- [69] Christodoulakis M, Iliopoulos CS, Park K, Sim JS. Implementing Approximate Regularities. *Mathematical and Computer Modelling*, 2005. **42**:855–866. doi:<https://doi.org/10.1016/j.mcm.2005.09.013>.
- [70] Amir A, Levy A, Lubin R, Porat E. Approximate Cover of Strings. *Theoret. Comput. Sci.*, 2019. **793**:59–69. doi:<https://doi.org/10.1016/j.tcs.2019.05.020>.
- [71] Amir A, Levy A, Lewenstein M, Lubin R, Porat B. Can We Recover the Cover? *Algorithmica*, 2019. **81**:2857–2875. doi:<https://doi.org/10.1007/s00453-019-00559-8>.
- [72] Amir A, Levy A, Porat E. Quasi-Periodicity Under Mismatch Errors. In: Proc. 29th Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 105. 2018 pp. 4:1–4:15. doi:10.4230/LIPIcs.CPM.2018.4.
- [73] Guth O. On Approximate Enhanced Covers under Hamming Distance. *Discrete Appl. Math.*, 2020. **274**:67–80. doi:<https://doi.org/10.1016/j.dam.2019.01.015>.
- [74] Guth O, Melichar B, Balik M. Searching All Approximate Covers and their Distance Using Finite Automata. In: Proc. Inf. Technologies — Appls. and Theory. 2008 pp. 21–26. URL <https://ceur-ws.org/Vol-414/>.
- [75] Guth O. Computing All Approximate Enhanced Covers with the Hamming Distance. *Proc. 20th Prague Stringology Conference (PSC)*, 2016. pp. 146–157. URL <http://www.stringology.org/event/2016/index.html>.
- [76] Kedzierski A, Radoszewski J.  $k$ -Approximate Quasiperiodicity under Hamming and Edit Distance. In: Proc. 31st Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 161. 2020 pp. 18:1–18:15. doi:10.4230/LIPIcs.CPM.2020.18.
- [77] Kedzierski A, Radoszewski J.  $k$ -Approximate Quasiperiodicity under Hamming and Edit Distance. *Algorithmica*, 2022. pp. 566—589. doi:<https://doi.org/10.1007/s00453-021-00842-7>.
- [78] Iliopoulos CS, Korda M. Optimal parallel superprimitivity testing on square arrays. *Parallel Processing Letters*, 1996. **6**(3):299–308. doi:<https://doi.org/10.1142/S0129626496000297>.
- [79] Crochemore M, Iliopoulos CS, Korda M. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 1998. **20**:353–373. doi:<https://doi.org/10.1007/PL00009200>.
- [80] Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 1975. **18**(6):333–340. doi:10.1145/360825.360855.

- [81] Popa A, Tanasescu A. An Output-Sensitive Algorithm for the Minimization of 2-Dimensional String Covers. In: Proc. 15th International Conference on Theory and Applications of Models of Computation (TAMC), volume LNCS 11436. 2019 pp. 536–549. doi:[https://doi.org/10.1007/978-3-030-14812-6\\_33](https://doi.org/10.1007/978-3-030-14812-6_33).
- [82] Charalampopoulos P, Radoszewski J, Rytter W, Waleń T, Zuba W. Computing Covers of 2D Strings. In: Proc. 32nd Annual Symp. Combinatorial Pattern Matching (CPM). 2021 pp. 12:1–12:20. doi:10.4230/LIPIcs.CPM.2021.12.
- [83] Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. Rectangular Tile Covers of 2D-Strings. In: Proc. 33rd Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 223. 2022 pp. 23:1–23:14. doi:10.4230/LIPIcs.CPM.2022.23.
- [84] Alzamel M, Conte A, Denzumi S, Grossi R, Iliopoulos CS, Kurita K, Wasa K. Finding the Anticover of a String. In: Proc. 31st Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 161. 2020 pp. 2.1–2.11. doi:10.4230/LIPIcs.CPM.2020.2.
- [85] Amir A, Boneh I, Kondratovsky E. Approximating the Anticover of a String. In: Proc. 27th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 12303. 2020 pp. 99–114. doi:[https://doi.org/10.1007/978-3-030-59212-7\\_8](https://doi.org/10.1007/978-3-030-59212-7_8).
- [86] Matsuda S, Inenaga S, Bannai H, Takeda M. Computing Abelian Covers and Abelian Runs. *Proc. 18th Prague Stringology Conference (PSC)*, 2014. pp. 43–51. URL <http://www.stringology.org/papers/PSC2014.pdf>.
- [87] Grossi R, Iliopoulos CS, Jansson J, Lim Z, Sung WK, Zuba W. Finding the Cyclic Covers of a String. In: Proc. 17th International Workshop on Algorithms & Computation (WALCOM), volume LNCS 13973. 2023 pp. 139–150. doi:[https://doi.org/10.1007/978-3-031-27051-2\\_13](https://doi.org/10.1007/978-3-031-27051-2_13).
- [88] Iliopoulos C, Kociumaka T, Radoszewski J, Rytter W, Waleń T, Zuba W. Linear Time Computation of Cyclic Roots and Cyclic Covers of a String. In: Proc. 34th Annual Symp. Combinatorial Pattern Matching (CPM). 2023 .
- [89] IT, Sugimoto S, Inenaga S, Bannai H, Takeda M. Computing Palindromic Factorizations and Palindromic Covers On-line. In: Proc. 25th Annual Symp. Combinatorial Pattern Matching (CPM), volume LNCS 8486. 2014 pp. 150–161. doi:[https://doi.org/10.1007/978-3-319-07566-2\\_16](https://doi.org/10.1007/978-3-319-07566-2_16).
- [90] Radoszewski J, Rytter W, Straszyński J, Waleń T, Zuba W. String Covers of a Tree. In: Proc. 28th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 12944. 2021 pp. 68–82. doi:[https://doi.org/10.1007/978-3-030-86692-1\\_7](https://doi.org/10.1007/978-3-030-86692-1_7).
- [91] Charalampopoulos P, Pissis SP, Radoszewski J, Rytter W, Waleń T, Zuba W. Subsequence Covers of Words. In: Proc. 29th String Processing & Inform. Retrieval Symp. (SPIRE), volume LNCS 13617. 2022 pp. 3–15. doi:[https://doi.org/10.1007/978-3-031-20643-6\\_1](https://doi.org/10.1007/978-3-031-20643-6_1).
- [92] Iliopoulos CS, Mohammed M, Mouchard L, Perdikuri KG, Smyth WF, Tsakalidis AK. String Regularities with Don't Cares. *Proc. 7th Prague Stringology Conference (PSC)*, 2002. pp. 65–74. URL <http://www.stringology.org/event/2002/index.html>.
- [93] Bari MF, Rahman MS, Shahriyar R. Finding all covers of an indeterminate string in  $O(n)$  time on average. In: Proc. 13th Prague Stringology Conference (PSC). 2009 pp. 263–271. URL <http://www.stringology.org/event/2009/index.html>.
- [94] Iliopoulos CS, Mohamed M, Mouchard L, Perdikuri K, Smyth WF, Tsakalidis AK. String Regularities with Don't Cares. *Nordic J. Comput.*, 2003. **10**(1):40–51.

- [95] Holub J, Smyth WF. Algorithms on indeterminate strings. In: Proc. 14th Australasian Workshop on Combinatorial Algs. (AWOCA). 2003 pp. 36–45.
- [96] Antoniou P, Crochemore M, Iliopoulos CS, Jayasekera I, Landau GM. Conservative String Covering of Indeterminate Strings. In: Proc. 12th Prague Stringology Conference (PSC). 2008 pp. 108–115. URL <http://www.stringology.org/event/2008/index.html>.
- [97] Antoniou P, Iliopoulos CS, Jayasekera I, Rytter W. Computing repetitive structures in indeterminate strings. In: Proc. 3rd Int'l Conference on Pattern Recognition in Bioinformatics (PRIB), volume LNCS 5265. 2008 URL <https://api.semanticscholar.org/CorpusID:8858579>.
- [98] Crochemore M, Iliopoulos CS, Kociumaka T, Radoszewski J, Rytter W, Walen T. Covering problems for partial words and for indeterminate strings. *Theoret. Comput. Sci.*, 2017. **698**:25–39. doi:<https://doi.org/10.1016/j.tcs.2017.05.026>.
- [99] Zhang H, Guo Q, Iliopoulos CS. Varieties of Regularities in Weighted Sequences. In: Proc. 6th International Conf. on Algorithmic Aspects in Information and Management, volume LNCS 7124. 2010 pp. 271–280. doi:[https://doi.org/10.1007/978-3-642-14355-7\\_28](https://doi.org/10.1007/978-3-642-14355-7_28).
- [100] Iliopoulos CS, Makris C, Panagis Y, Perdikuri K, Theodoridis E, Tsakalidis A. The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications. *Fundamenta Informaticae*, 2006. **71**:259–277.
- [101] Barton C, Kociumaka T, Lie C, Pissis SP, Radoszewski J. Indexing Weighted Sequences: Neat and Efficient. *Information and Computation*, 2019. **270**:104462.1–104462.21. doi:10.1016/j.ic.2019.104462.
- [102] Berkman O, Iliopoulos CS, Park K. The Subtree Max Gap Problem with Application to parallel String Covering. *Information and Computation*, 1995. **123**:127–137. doi:10.1006/inco.1995.1162.
- [103] Smyth W. Repetitive perhaps, but certainly not boring. *Theoretical Computer Science*, 2000. **246**:343–355. doi:[https://doi.org/10.1016/S0304-3975\(00\)00067-0](https://doi.org/10.1016/S0304-3975(00)00067-0).
- [104] Kociumaka T, Kubica M, Radoszewski J, Rytter W, Walen T. A Linear-Time Algorithm for Seeds Computation. In: Proc. 23rd CM–SIAM Symp. on Discrete Algorithms (SODA). 2012 pp. 1095–1112.
- [105] Christou M, Crochemore M, Guth O, Iliopoulos CS, Pissis SP. On the left and right seeds of a string. *J. Discrete Algorithms*, 2012. **17**:31–44. doi:<https://doi.org/10.1016/j.jda.2012.10.004>.
- [106] Christou M, Crochemore M, Iliopoulos CS, Kubica M, Pissis SP, Radoszewski J, Rytter W, Szreder B, Walen T. Efficient seeds computation revisited. In: Proc. 22nd Annual Symp. Combinatorial Pattern Matching (CPM), volume LNCS 6661. 2011 pp. 350–363. doi:[https://doi.org/10.1007/978-3-642-21458-5\\_30](https://doi.org/10.1007/978-3-642-21458-5_30).
- [107] Christou M, Crochemore M, Guth O, Iliopoulos CS, Pissis SP. On the right-seed array of a string. In: Proc. 17th Annual International Computing & Combinatorics Conference (COCOON). 2012 pp. 492–502. doi:[https://doi.org/10.1007/978-3-642-22685-4\\_43](https://doi.org/10.1007/978-3-642-22685-4_43).
- [108] Crochemore M. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 1981. **12**(5):244–250. doi:[https://doi.org/10.1016/0020-0190\(81\)90024-7](https://doi.org/10.1016/0020-0190(81)90024-7).
- [109] Guo Q, Zhang H, Iliopoulos CS. Computing the  $\lambda$ -Seeds of a String. In: Proc. 2nd International Conf. on Algorithmic Aspects in Information and Management, volume LNCS 4041. 2006 pp. 303–313. doi:10.1007/11775096\_28.
- [110] Christodoulakis M, Iliopoulos CS, Park K, Sim JS. Approximate Seeds of Strings. *J. Automata, Languages & Combinatorics*, 2005. **10**(5/6):609–626. doi:10.25596/jalc-2005-609.

- [111] Maier D. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*, 1978. **25**(2):322–336. doi:<https://doi.org/10.1145/322063.322075>.
- [112] Rähä K, Ukkonen E. The Shortest Common Supersequence Problem over Binary Alphabet is NP-Complete. *Theoret. Comput. Sci.*, 1981. **16**:187–198. doi:10.1016/0304-3975(81)90075-X.
- [113] Guth O, Melichar B. Using Finite Automata Approach for Searching Approximate Seeds of Strings. In: Proc. Intelligent Automation and Computer Engineering, volume LNCS 52. 2010 pp. 347–360. doi:10.1007/978-90-481-3517-2-27.
- [114] Guth O, Melichar B. Searching All Seeds of Strings with Hamming Distance using Finite Automata. In: Proc. International MultiConference of Engineers and Computer Scientists (IMECS), volume 1. 2009 URL <https://www.iaeng.org/publication/IMECS2009/>.
- [115] Guth O, Melichar B. Finite Automata Approach to Computing All Seeds of Strings with the Smallest Hamming Distance. *IAENG International Journal of Computer Science*, 2009. **36**(2). URL [https://www.iaeng.org/IJCS/issues\\_v36/issue\\_2/](https://www.iaeng.org/IJCS/issues_v36/issue_2/).
- [116] Guth O. Searching Regularities in Strings using Finite Automata. Ph.D. thesis, Czech Technical University in Prague, Zikova 1903/2, Praha, CZ 160 00, 2014.
- [117] Kociumaka T, Pissis SP, Radoszewski J, Rytter W, Walen T. Efficient algorithms for shortest partial seeds in words. *Theoret. Comput. Sci.*, 2018. **710**:139–147. doi:<https://doi.org/10.1016/j.tcs.2016.11.035>.
- [118] Gawrychowski P, Radoszewski J, Starikovskaya T. Quasi-Periodicity in Streams. In: Proc. 30th Annual Symp. Combinatorial Pattern Matching (CPM), volume LIPIcs 128. 2019 pp. 22:1–22:14. doi:10.4230/LIPIcs.CPM.2019.22.
- [119] Kociumaka T, Radoszewski J, Wiśniewski B. Subquadratic-Time Algorithms for Abelian Stringology Problems. In: Proc. 6th Mathematical Aspects of Computer and Information Sciences (MACIS), volume LNCS 9582. 2015 doi:[https://doi.org/10.1007/978-3-319-32859-1\\_27](https://doi.org/10.1007/978-3-319-32859-1_27).
- [120] Radoszewski J, Rytter W, Straszypiński J, Walen T, Zuba W. Hardness of Detecting Abelian and Additive Square Factors in Strings. In: Proc. 29th Annual European Symposium on Algorithms (ESA), volume LIPIcs 2021. 2021 pp. 77:1–77:19. doi:10.4230/LIPIcs.ESA.2021.77.