SIAM J. COMPUT.
Vol. 53, No. 5, pp. 1524–1577

INTERNAL PATTERN MATCHING QUERIES IN A TEXT AND APPLICATIONS*

TOMASZ KOCIUMAKA†, JAKUB RADOSZEWSKI‡, WOJCIECH RYTTER§, AND TOMASZ WALEŃ§

Abstract. We consider several types of internal queries, that is, questions about *fragments* of a given text T specified in constant space by their locations in T. Our main result is an optimal data structure for internal pattern matching (IPM) queries, which, given two fragments x and y, ask for a representation of all fragments contained in y and matching x exactly. This problem can be viewed as an internal version of the fundamental exact pattern matching problem: We are looking for exact occurrences of one substring of T within another substring of T. Our data structure answers IPM queries in time proportional to the quotient $|y|/|x|$ of the fragments' lengths, which is required due to the worst-case information content of the output. If T is a text of length n over an integer alphabet of size σ, then our data structure occupies $\mathcal{O}(n/\log_\sigma n)$ machine words (that is, $\mathcal{O}(n\log\sigma)$ bits) and admits an $\mathcal{O}(n/\log_\sigma n)$-time construction algorithm. We also show how to use IPM queries for answering internal queries corresponding to other classic string processing problems. Among others, we derive optimal data structures reporting the periods of a fragment and testing the cyclic equivalence of two fragments. Since the publication of the conference version of this paper [Kociumaka et al., *Internal pattern matching queries in a text and applications*, SODA 2015], IPM queries have found numerous further applications, following the path paved by the classic longest common extension (LCE) queries of Landau and Vishkin [*J. Comput. System Sci.*, 37 (1988), pp. 63–78]. In particular, IPM queries have been implemented in grammar-compressed and dynamic settings and, along with LCE queries, constitute elementary operations of the `PILLAR` model, developed by Charalampopoulos, Kociumaka, and Wellnitz [*Faster approximate pattern matching: A unified approach*, FOCS 2020] to design approximate pattern matching algorithms that work in multiple settings. All our algorithms are deterministic, whereas the data structure in the conference version of the paper only admits a randomized construction in $\mathcal{O}(n)$ expected time. To achieve this, we provide a novel construction of *string synchronizing sets* of Kempa and Kociumaka [*String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure*, STOC 2019]. Our method, based on a new *restricted* version of the recompression technique of Jeż [*J. ACM*, 63 (2016), pp. 4:1–4:51], yields a hierarchy of $\mathcal{O}(\log n)$ string synchronizing sets covering the whole spectrum of the fragments' lengths.

Key words. pattern matching, internal queries, synchronizing sets, data structures, local consistency

MSC codes. 68W32, 68P05, 68W05, 68Q25

DOI. 10.1137/23M1567618

1. Introduction.

In this paper, we consider *internal queries*, which ask to solve instances of a certain string-processing problem with input strings given as fragments of a fixed string T represented by their endpoints. The task is to preprocess T, called the *text*, into a data structure that efficiently answers certain types of internal queries. In retrospect, the origins of internal queries in texts can be traced back to the work

*Received by the editors May 2, 2023; accepted for publication (in revised form) July 3, 2024; published electronically October 14, 2024. A preliminary version of this work was presented at the 26th Annual ACM-SIAM Symposium on Discrete Algorithms [76] and included in the Ph.D. thesis of the first author [71].
https://doi.org/10.1137/23M1567618
Funding: The work of the second author was supported by the Polish National Science Center (grant 2022/46/E/ST6/004).
†Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany (tomasz.kociumaka@mpi-inf.mpg.de).
‡University of Warsaw and Samsung R&D Warsaw, Warsaw, Poland (jrad@mimuw.edu.pl).
§University of Warsaw, Warsaw, Poland (rytter@mimuw.edu.pl, walen@mimuw.edu.pl).

of Landau and Vishkin [82] introducing Longest Common Extension Queries (LCE Queries). We develop data structures with efficient (in most cases optimal) deterministic construction and query algorithms for internal versions of several natural problems in string processing.

We always denote the input text by T and its length by n. We also make a standard assumption (cf. [43]) that the characters of T are (or can be identified with) integers $[0\,..\,\sigma)$,[1] where $\sigma = n^{\mathcal{O}(1)}$; that is, T is over a polynomially bounded *integer alphabet.* Our results are designed for the standard word RAM model with machine words of $\omega \geq \log n$ bits.[2] In this model, the text T can be represented using $\mathcal{O}(n/\log_\sigma n)$ machine words, that is, $\mathcal{O}(n \log \sigma)$ bits, in a so-called packed representation; see [18]. We consider data structures that use $\mathcal{O}(n/\log_\sigma n)$ space and can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time. For example, Kempa and Kociumaka [71, 65] showed that such space complexity and preprocessing time are sufficient for constant-time LCE Queries.

We consider the fundamental exact pattern matching problem [91, 55, 111, 3, 20, 69, 63] in the following internal version, which asks for the exact occurrences of one fragment within another fragment; see Figure 1. By $|w|$, we denote the length of a string or fragment w.

Internal Pattern Matching (IPM) Queries
Given fragments x and y of the text T satisfying $|y| < 2|x|$, report the fragments matching x and contained in y (represented as an arithmetic progression of their starting positions).

We impose the restriction $|y| < 2|x|$ so that the output can be represented in constant space: In this case, the starting positions of the occurrences of x in y form an arithmetic progression; see [21, 96]. If $|y| \geq 2|x|$, then one can ask $\mathcal{O}(|y|/|x|)$ IPM Queries (to find the occurrences of x within fragments of length $2|x| - 1$ contained in y, with overlaps of at least $|x| - 1$ characters between the subsequent fragments) and output $\mathcal{O}(|y|/|x|)$ arithmetic progressions. As shown in Appendix B,

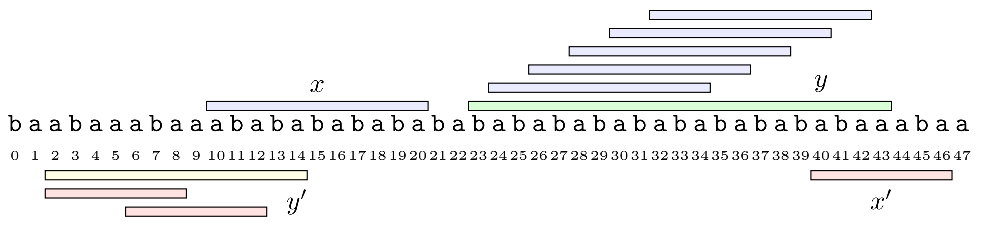


Fig. 1. *Two* IPM Queries *on a text* T*. The fragment* $x = T[10\,..\,21)$ *has five occurrences in the fragment* $y = T[23\,..\,44)$*; their starting positions form an arithmetic progression* $24, 26, 28, 30, 32$ *that can be represented uniquely by three integers* $(24, 26, 32)$*. Fragment* $x' = T[40\,..\,47)$ *has two occurrences in the fragment* $y' = T[2\,..\,15)$*; their starting positions,* 2 *and* 6*, also form an arithmetic progression.*

[1]For $i, j \in \mathbb{Z}$, we denote $[i\,..\,j] = \{k \in \mathbb{Z} : i \leq k \leq j\}$, $[i\,..\,j) = \{k \in \mathbb{Z} : i \leq k < j\}$, and $(i\,..\,j] = \{k \in \mathbb{Z} : i < k \leq j\}$.

[2]Throughout this paper, log denotes the base-2 logarithm (any other base is explicitly provided in the subscript).

this representation is optimal from the information-theoretic perspective, and thus the $\mathcal{O}(|y|/|x|)$-factor overhead is necessary.

Remark 1.1. For $|y| < 2|x|$, answering IPM queries is equivalent to computing the leftmost occurrence of x in y. Such a subroutine also allows finding the second occurrence from the left (if y is trimmed appropriately to avoid the leftmost occurrence) as well as the rightmost occurrence (if the text is reversed).

We design an optimal data structure for IPM Queries.

Theorem 1.2 (main result). *For every text $T \in [0\,..\,\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* IPM Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Linear-time solutions of the exact pattern matching problem [91, 69, 20] are among the foundational results of string algorithms. Although this alone can motivate the study of IPM Queries, the significance of our result is primarily due to a growing collection of applications. In this work, we show that several further internal queries can be answered efficiently using Theorem 1.2; these are described in detail in section 1.1. Other applications of IPM Queries, developed after the conference version of this paper [76], are listed in section 1.2.

Outline of the solution. A key technical contribution behind Theorem 1.2 is a novel construction of *string synchronizing sets* of Kempa and Kociumaka [65]. Intuitively, a string synchronizing set corresponds to a set of relatively few fragments of a specified length selected in a locally consistent way. Kempa and Kociumaka [65] showed a linear-time algorithm constructing a single synchronizing set of asymptotically optimum size for fragments with specified lengths. We generalize this result to obtain, still in linear time and space, a *synchronizing sets hierarchy* consisting of $\mathcal{O}(\log n)$ string synchronizing sets covering the whole spectrum of the fragments' lengths. As opposed to [65], our construction does not use sliding-window minima (known as *minimizers* [101]), but we rely on locally consistent parsing, a concept dating back to the mid-1990s [103, 104, 105, 88]. Specifically, we adapt the *recompression* technique of Jeż [59, 60], which results in a simple and efficient parsing scheme.

The synchronizing sets hierarchy is obtained using a modified version of recompression and properties of maximal repetitions (called *runs*) in the text. This hierarchy and the structure of *highly periodic runs* are essentially independent basic components of our main data structure. Queries with highly periodic patterns are answered using runs, whereas other queries rely on the synchronizing sets hierarchy.

Structure of the paper. The main technical contributions of the paper are listed in section 1.4. Properties of periodicities and runs are discussed in section 2. Section 3 gives a warm-up for more complicated sections 4 and 5, containing technical details of synchronizing hierarchies and recompression. In the subsequent sections, the synchronizing sets hierarchy together with runs is used as the main data structure for IPM Queries (with $\mathcal{O}(n)$-time preprocessing in sections 6 and 7 and with $\mathcal{O}(n/\log_\sigma n)$-time preprocessing in section 8). The last section (section 9) covers the applications listed in section 1.1.

1.1. Applications of IPM queries.

Period queries. One of the central notions of combinatorics on words is that of a *period* of a string. An integer $p \in [1\,..\,|w|]$ is a *period* of a string w if $w[i] = w[i+p]$ holds for all $i \in [0\,..\,|w|-p)$. Already, Morris and Pratt [91] and Knuth, Morris, and

Pratt [69] provided a linear-time procedure listing all periods of a given string (as a side result of their linear-time pattern matching algorithm).

The sorted sequence of periods of a length-m string can be cut into $\mathcal{O}(\log m)$ arithmetic progressions [69]. A complete characterization of the possible families of periods [53] further shows that the size of such a representation ($\Theta(\log^2 m)$ bits) is asymptotically tight.[3] Hence, we adopt it in the internal version of the problem of finding all periods of a string, formally specified below.

PERIOD QUERIES
Given a fragment x of T, report all periods of x (represented by disjoint arithmetic progressions).

We have introduced PERIOD QUERIES in [75], presenting two solutions. The first data structure takes $\mathcal{O}(n \log n)$ space and answers PERIOD QUERIES in the optimal $\mathcal{O}(\log |x|)$ time after $\mathcal{O}(n \log n)$-time randomized construction. The other one is based on orthogonal range searching; its size is $\mathcal{O}(n + S_{rsucc}(n))$, and the query time is $\mathcal{O}(Q_{rsucc}(n) \cdot \log |x|)$, where $S_{rsucc}(n)$ and $Q_{rsucc}(n)$ are analogous quantities for data structures answering range successor queries; see section 9.4 for a definition. The state-of-the-art trade-offs are $S_{rsucc}(n) = \mathcal{O}(n)$ and $Q_{rsucc}(n) = \mathcal{O}(\log^{\varepsilon} n)$ for every constant $\varepsilon > 0$ [93], $S_{rsucc}(n) = \mathcal{O}(n \log\log n)$ and $Q_{rsucc}(n) = \mathcal{O}(\log\log n)$ [113], as well as $S_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ and $Q_{rsucc}(n) = \mathcal{O}(1)$ for every constant $\varepsilon > 0$ [37]. The first two of these data structures can be constructed in time $C_{rsucc}(n) = \mathcal{O}(n\sqrt{\log n})$ [17, 48], whereas the third one can be constructed in time $C_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ [37].

In this paper, we develop an asymptotically optimal data structure answering PERIOD QUERIES.

THEOREM 1.3. *For every text $T \in [0\,..\,\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* PERIOD QUERIES *in $\mathcal{O}(\log |x|)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Our query algorithm is based on the intrinsic relation between periods and *borders* (i.e., substrings occurring both as prefixes and as suffixes) of a string. In fact, to answer each PERIOD QUERY, it combines the results of the following PREFIX-SUFFIX QUERIES used with $x = y$ to determine the borders of x.

PREFIX-SUFFIX QUERIES
Given fragments x and y of T and a positive integer d, report all suffixes of y of length in $[d\,..\,2d)$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths).

In other words, we prove the following auxiliary result.

THEOREM 1.4. *For every text $T \in [0\,..\,\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* PREFIX-SUFFIX QUERIES *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

[3]Recently, $(\frac{1}{2} \pm o(1)) \log^2 m$ bits were proved to be sufficient and, in the worst case, necessary to encode the set of all periods of a length-m string [99, 100].

In many scenarios, very long periods ($p = m - o(m)$ for a string of length m) are irrelevant. The remaining periods correspond to borders of length $\Theta(m)$ and thus can be retrieved with just a constant number of Prefix-Suffix Queries. The case of $p \le \frac{1}{2}m$ is especially important since fragments x with periods not exceeding $\frac{1}{2}|x|$, called *periodic* fragments, can be uniquely extended to *maximal repetitions*, also known as *runs* (see section 2). We denote the unique run extending a periodic fragment x by $\mathsf{run}(x)$. If x is not periodic, we leave $\mathsf{run}(x)$ undefined, which we denote as $\mathsf{run}(x) = \perp$.

Periodic Extension Queries
Given a fragment x of T, compute the run $\mathsf{run}(x)$ that extends x.

Theorem 1.4 along with the optimal data structure for LCE Queries [65] imply the following result.

Theorem 1.5. *For every text $T \in [0\,..\,\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* Periodic Extension Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Bannai et al. [16] presented an alternative implementation of Periodic Extension Queries with $\mathcal{O}(n)$-time construction and $\mathcal{O}(1)$-time queries. The underlying special case of Period Queries also generalizes Primitivity Queries (asking if a fragment x is *primitive,* i.e., whether it does not match u^k for any string u and integer $k \ge 2$), earlier considered by Crochemore et al. [36], who developed a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ with an $\mathcal{O}(Q_{rsucc}(n))$-time query algorithm.

Cyclic equivalence queries. Consider a *rotation* operation that moves the last character of a given string u to the front. Formally, if $|u| = m$, then $\mathsf{rot}(u) = u[m-1]u[0]\cdots u[m-2]$. In general, for $j \in \mathbb{Z}$, we define the rot^j function as the jth function power of rot. Two strings u and v are called cyclically equivalent if $u = \mathsf{rot}^j(v)$ holds for some integer j. A classic linear-time algorithm for checking cyclic equivalence of strings u and v performs pattern matching for u in v^2 [87]; there is also a simple linear-time constant-space algorithm [39]. We define the following queries.

Cyclic Equivalence Queries
Given two fragments x and y of T, return $\{j \in \mathbb{Z} : \mathsf{rot}^j(x) = y\}$ (represented as an arithmetic progression).

Theorem 1.6. *For every text $T \in [0\,..\,\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* Cyclic Equivalence Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Let us mention that [70] gave an alternative data structure answering Cyclic Equivalence Queries in constant time after $\mathcal{O}(n)$-time preprocessing; this solution also supports constant-time queries asking for the lexicographically minimal cyclic rotation of a given fragment.

Bounded LCP queries. Keller et al. [64] used the following queries in the solution to their Generalized LZ Substring Compression Queries problem.

Bounded Longest Common Prefix (LCP) Queries
Given two fragments x and y of T, find the longest prefix p of x that occurs in y.

Our result for IPM Queries can be combined with the techniques of [64] in a more efficient implementation of Bounded LCP Queries. Compared to the original version, the resulting data structure, specified below, has a $\log\log|p|$ factor instead of a $\log|p|$ factor in the query time.

Theorem 1.7. *For every text T of length n over an alphabet $[0\mathinner{.\,.} n^{\mathcal{O}(1)})$, there exists a data structure of size $\mathcal{O}(n+S_{rsucc}(n))$ that answers* Bounded LCP Queries *in $\mathcal{O}(Q_{rsucc}(n)\log\log|p|)$ time. The data structure can be constructed in $\mathcal{O}(n+C_{rsucc}(n))$ time.*

In section 9, we formally define Generalized LZ Substring Compression Queries and discuss how Theorem 1.7 lets us improve and extend the results of [64].

Earlier versions of our results. Weaker versions of Theorems 1.2–1.6 (with $\mathcal{O}(n)$ space and construction time) and Theorem 1.7 were published in the conference version of the paper [76] and the Ph.D. thesis of the first author [71]. Moreover, the construction algorithms provided in [76] were Las Vegas randomized, with linear bounds on the *expected* construction time only.

1.2. Further applications of IPM queries. Since the publication of the conference version of this work [76], the list of internal queries, predominantly implemented using IPM Queries, has grown to include shortest unique substrings [2], longest common substring [10], suffix rank and selection [14, 70], BWT substring compression [14], shortest absent string [15], dictionary matching [25], string covers [38], masked prefix sums [41], circular pattern matching [57], and longest palindrome [89].

Furthermore, IPM Queries have been used in efficient algorithms for many problems, such as approximate pattern matching [30, 31], approximate circular pattern matching [27, 28], RNA folding [40], and computing string covers [98]. Additionally, IPM Queries have found further indirect applications that are based on the internal queries from section 1.1: Periodic Extension Queries have been applied for approximate period recovery [6, 9], dynamic repetition detection [8], identifying two-dimensional maximal repetitions [12], enumeration of distinct substrings [29], and pattern matching with variables [80, 45], whereas Prefix-Suffix Queries have been applied for detecting gapped repeats and subrepetitions [77, 50], in the dynamic longest common substring problem [10], and for computing the longest unbordered substring [72].

The fundamental role of IPM Queries as a building block for the design of string algorithms motivated their efficient implementation in the compressed and dynamic settings [30, 66, 67]. In particular, the `PILLAR` model, introduced in [30] with the aim of unifying approximate pattern matching algorithms across different settings, includes IPM Queries as one of the primitives. It also includes LCE Queries [82], defined as $\mathrm{LCE}(i,i')=\mathrm{lcp}(w[i\mathinner{.\,.}|w|),w[i'\mathinner{.\,.}|w|))$, where $\mathrm{lcp}(v,w)$ denotes the longest common prefix of two strings v,w, and LCE Queries on reversed strings, as well as the following basic primitives:

- `Extract`(w,ℓ,r): Given a string w and integers $0\le\ell\le r\le|w|$, retrieve the string $w[\ell\mathinner{.\,.} r)$.

- `Access`(w,i): Given a string w and a position $i \in [0\,.\,.\,|w|)$, retrieve the character $w[i]$.
- `Length`(w): Retrieve the length $|w|$ of the string w.

The argument strings of `PILLAR` primitives are represented as fragments of one or more strings in a given text collection $\mathcal{X}$.

Using an earlier version of Theorem 1.2, providing $\mathcal{O}(n)$-time deterministic construction [71], it has been observed [30, Theorem 7.2] that, after $\mathcal{O}(n)$-time preprocessing of a collection $\mathcal{X}$ of strings of total length n, each `PILLAR` operation can be performed in $\mathcal{O}(1)$ time. We improve on this result using Theorem 1.2 to implement IPM Queries and the following implementation of LCE Queries.

Proposition 1.8 ([65]). *For every text T of length n over alphabet $[0\,.\,.\,\sigma)$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* LCE Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

We apply elementary bitwise operations for `Access` queries (the `Extract` and `Length` queries are straightforward since $w = X[\ell\,.\,.\,r)$ is represented by a pointer to $X \in \mathcal{X}$ and the two endpoints ℓ and r). This gives the following result.

Theorem 1.9. *A collection $\mathcal{X}$ of strings of total length n over alphabet $[0\,.\,.\,\sigma)$ can be preprocessed in $\mathcal{O}(|\mathcal{X}| + n/\log_\sigma n)$ time so that each* `PILLAR` *operation on strings from $\mathcal{X}$ can be performed in $\mathcal{O}(1)$ time.*

Since the approximate pattern matching algorithms of [30, 31, 28, 32] are implemented in the `PILLAR` model, Theorem 1.9 immediately improves their running times for strings over small alphabets. In particular, given a pattern $p \in [0\,.\,.\,\sigma)^m$, a text $t \in [0\,.\,.\,\sigma)^n$, and a threshold k, the occurrences of p in t with at most k mismatches (substitutions) can be reported in time $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot \min(k\sqrt{m\log m}, k^2))$, whereas the occurrences with at most k edits (insertions, deletions, and substitutions) can be reported in time $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot k^{3.5}\sqrt{\log m \log k})$. In both cases, the improvement is that the $\mathcal{O}(n)$ term, dominating the complexity of previous state-of-the-art solutions [24, 31] for small values of k, is replaced by $\mathcal{O}(n/\log_\sigma n)$. A similar phenomenon applies to circular pattern matching with at most k mismatches [28], where we achieve $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot k^3 \log\log k)$ time for the reporting version of the problem and $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot k^2 \log k/\log\log k)$ for the decision version, and to circular pattern matching with at most k edits [32], where we achieve $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot k^6)$ time for the reporting version of the problem and $\mathcal{O}(n/\log_\sigma n + (n/m)\cdot k^5 \log^3 k)$ for the decision version.

We also note that the query algorithms behind Theorems 1.3–1.6 access the text only through `PILLAR` operations. Thus, we obtain the following result.

Corollary 1.10. *Let $x, y \in \mathcal{X}$ be strings of length at most n. In the* `PILLAR` *model, one can compute the following:*

1. *(*Prefix-Suffix Query*) for a given positive integer d, all suffixes of y of length in $[d\,.\,.\,2d)$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths) in $\mathcal{O}(1)$ time;*
2. *(*Period Query*) all periods of x (represented by disjoint arithmetic progressions) in $\mathcal{O}(\log n)$ time;*
3. *(*2-Period Query*) the smallest period of x provided that x is periodic in $\mathcal{O}(1)$ time;*
4. *(*Cyclic Equivalence Query*) the set $\{j \in \mathbb{Z} : \mathsf{rot}^j(x) = y\}$ (represented as an arithmetic progression) in $\mathcal{O}(1)$ time.*

With known implementations of the PILLAR model, we automatically obtain efficient implementations of the queries of Corollary 1.10 in the dynamic, fully compressed, and quantum settings; see Appendix C.

1.3. Related queries. Internal queries are not the only problems in the literature involving fragments of a static text. Other variants include interval LCP queries [34, 64], range LCP queries [94, 7, 13, 1], substring hashing queries [44, 49, 52], fragmented pattern matching queries [11, 52], and cross-document pattern matching queries [79], to mention a few. In particular, interval LCP queries can be used to solve the decision version of IPM Queries. Keller et al. [64] showed how to answer the decision version of IPM Queries in $\mathcal{O}(Q_{rsucc}(n))$ time using a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time. The aforementioned query time is valid for arbitrary lengths $|x|$ and $|y|$, so the efficiency of this data structure is incomparable to our Theorem 1.2.

The setting of internal queries does not include problems like text indexing, where the text is available in advance but the pattern is explicitly provided at query time. This restricts the expressibility of internal queries but, at the same time, allows for greatly superior query times, which do not need to account for reading any strings. Another difference is in the typical usage scenario: Data structures for indexing problems are primarily designed to be interactively queried by a user. In other words, they are meant to be constructed relatively rarely, but they need to be stored for prolonged periods of time. As a result, space usage (including the multiplicative constants) is heavily optimized, whereas the efficiency of construction procedures is of secondary importance. On the other hand, internal queries often arise during bulk processing of textual data. The relevant data structures are then built within the preprocessing phase of the enclosing algorithms, so the running times of the construction procedures are counted toward the overall processing time. In this setting, efficient construction is as significant as fast queries.

1.4. Technical contributions. Below, we briefly introduce the most important technical contributions of our work. We start with a high-level overview of our IPM Queries data structure for large alphabets, that is, $\sigma = n^{\Theta(1)}$. In this setting, the space and construction time bounds of Theorem 1.2 simplify to $\mathcal{O}(n)$.

String matching using (deterministic) samples. The idea of (deterministic) sampling is a classic technique originally developed for parallel string matching algorithms [109] and later applied in other contexts, such as quantum string matching [54]. Algorithms using this approach to find the occurrences of a pattern x in a text y follow a three-phase scheme. In the preprocessing phase, a subset of characters of x, called the *sample*, is determined. Then, in the filtering phase, the algorithm selects all fragments $y[i \mathinner{.\,.} i+|x|)$ of y that match the sample. Finally, in the verification phase, the algorithm checks whether each candidate $y[i \mathinner{.\,.} i+|x|)$ matches the whole pattern x. The efficiency of this scheme hinges on two properties of the chosen samples: (1) They must be simple enough to support efficient filtering, and (2) they must carry enough information to leave few candidates for the verification phase. For example, Vishkin [109] chooses a sample of size $\mathcal{O}(\log |x|)$ so that the starting positions of the candidates surviving the filtering phase form $\mathcal{O}(|y|/|x|)$ arithmetic progressions whose difference is the smallest period of x.

Our data structure answering IPM Queries, described in section 6, uses contiguous samples; that is, the sample of a pattern x can be interpreted as a fragment of T contained in x. Moreover, we minimize the total number of samples across all the fragments of T rather than the size of each sample, and we aim to choose the samples *consistently* so that any fragment matching a sample is a sample itself. These constraints are infeasible for highly periodic (HP) patterns, whose length is much larger than the smallest period (we set $\frac{1}{3}|x|$ as the cutoff point for the period of x), and such patterns are handled in section 7 using different techniques outlined later on. As for non–highly periodic (NHP) patterns, we choose $\mathcal{O}(n)$ samples in total so that the smallest period of the sample of x is $\Omega(|x|)$. The small number of samples allows storing them explicitly, whereas the large period guarantees that the sample of x has $\mathcal{O}(1)$ occurrences within any fragment y of length $|y| < 2|x|$ (due to Fact 2.3). Thus, at query time, our data structure uses the precomputed set of samples (stored in appropriate deterministic dictionaries [102, 95]) to identify the sample of x and its occurrences in y. Then we use LCE Queries to test which occurrences of the sample extend to occurrences of x.

String synchronizing sets hierarchy. The challenge of implementing the strategy outlined above is to consistently pick $\mathcal{O}(n)$ samples among $\Theta(n^2)$ fragments of T. The natural first step is to restrict the selection to $\Theta(n \log n)$ fragments whose lengths form a geometric progression, but any further reduction in the number of samples requires nontrivial symmetry breaking.

In the conference version of this paper [76], we employed a strategy reminiscent of the *minimizers* technique popular in bioinformatics [101, 112, 42, 83] and known under different names in many other applications [106, 107, 23, 97]. In that approach, candidate fragments are consistently assigned uniformly random weights, and the sample of x is defined as the minimum-weight fragment of a certain length (such as $2^{\lfloor \log |x| \rfloor - 1}$) contained in x. With minor adaptations (necessary to avoid highly periodic samples), this scheme yields $\mathcal{O}(n/2^k)$ length-2^k samples in expectation for every $k \in [0 \mathinner{..} \lfloor \log n \rfloor]$. Subsequently, our sample selection algorithm was adapted for answering LCE Queries [19, 71] and for the Burrows–Wheeler transform construction [65]. The latter paper contributed a clean notion of a *string synchronizing set*, which has since been applied in many further contexts (see, e.g., [66, 26, 67, 61]).

Intuitively, for a length-n string T and a parameter $\tau \in [1 \mathinner{..} \lfloor \frac{n}{2} \rfloor]$, a τ-synchronizing set Sync is a subset of positions in T such that the decision to include a position in Sync depends only on characters at the subsequent 2τ positions (consistency) and, among every τ consecutive position, at least one is included in Sync unless the τ positions are located in a highly periodic fragment of T (density); see Definition 3.1 for a formal definition. Kempa and Kociumaka [65] obtained the following result building on our original sample-selection algorithm [76] and its derandomized version presented in [71].

Proposition 1.11 (see [65, Proposition 8.10 and Theorem 8.11]). *For every text $T \in [0 \mathinner{..} \sigma)^n$ with $\sigma = n^{\mathcal{O}(1)}$ and $\tau \in [1 \mathinner{..} \lfloor \frac{n}{2} \rfloor]$, there exists a τ-synchronizing set of size $\mathcal{O}(n/\tau)$ that can be constructed in $\mathcal{O}(n)$ time. Moreover, if $\tau \le \frac{1}{5} \log_\sigma n$ and T is given in a packed representation, then the construction time can be improved to $\mathcal{O}(n/\tau)$.*

It is not hard to argue (see Lemma 6.4) that one can pick as samples all the length-1 fragments as well as, for some fixed 2^{k-1}-synchronizing sets across all $k \in [1 \mathinner{..} \lfloor \log n \rfloor]$, all the 2^{k-1}-synchronizing fragments.[4] Unfortunately, it takes $\mathcal{O}(n \log n)$ time to construct these synchronizing sets using the algorithm of Proposition 1.11. Consequently, the deterministic version of Theorem 1.2 published in [71] follows a

[4]The τ-synchronizing fragments are length-2τ fragments starting at all positions contained in a fixed τ-synchronizing set.

sophisticated approach relying on just two string synchronizing sets, constructed in $\mathcal{O}(n)$ time each. In this paper, we apply a more natural strategy and show that the entire *hierarchy* of 2^{k-1}-synchronizing sets can be constructed in deterministic $\mathcal{O}(n)$ time, as captured in the definition and theorem below.

DEFINITION 1.12. *A* synchronizing sets hierarchy *of a text T of length n is a data structure that, given any $\tau \in [1\mathinner{.\,.}\lfloor \frac{n}{2} \rfloor]$, constructs a τ-synchronizing set* Sync *of T.*

THEOREM 1.13 (construction of synchronizing sets hierarchy). *Given a text T of length n over an alphabet $[0\mathinner{.\,.}n^{\mathcal{O}(1)})$, one can construct in $\mathcal{O}(n)$ time a synchronizing sets hierarchy that, for any $\tau \in [1\mathinner{.\,.}\lfloor \frac{n}{2} \rfloor]$, in $\mathcal{O}(\frac{n}{\tau})$ time returns a τ-synchronizing set* Sync *of T such that $|\mathsf{Sync}| < 70\,\frac{n}{\tau}$ and every τ-synchronizing fragment induced by* Sync *has smallest period larger than $\frac{\tau}{3}$.*

In the proof of Theorem 1.13, which is presented in section 5, we use the *restricted recompression* technique discussed below.

Restricted recompression. Locally consistent parsing algorithms [109, 88, 60] construct a hierarchical factorization of T, where level-0 phrases are single characters of T, the only level-q phrase (for some $q = \mathcal{O}(\log n)$) is the entire text T, and, for each $k \in [1\mathinner{.\,.}q]$, the level-$k$ phrases are concatenations of level-$(k-1)$ phrases. In this context, local consistency means that whether two subsequent level-$(k-1)$ phrases are merged into the same level-k phrase is a local decision that depends only on a few neighboring level-$(k-1)$ phrases. Unfortunately, the lengths of level-$(k-1)$ phrases can vary significantly between regions of the text, and thus it is impossible to ensure that the symmetry-breaking decisions are made based on fixed-size contexts, as required by the consistency property of string synchronizing sets.

Thus, we alter the original recompression algorithm and introduce a small but consequential restriction: Phrases deemed too long for their level are never merged with their neighbors. Although this trick does not eliminate very long phrases, it lets us quantify local consistency in terms of fixed-size contexts.

Our restricted recompression scheme has already been used in [74, 68] to prove that every text admits an efficiently constructible run-length straight-line program of a particular size. Restricted variants of other locally consistent parsing schemes have been used in [19, 73, 67] to derive efficient small-space, compressed, and dynamic text indexes.

In section 3, we provide further intuition and explain how to define synchronizing sets in terms of phrase boundaries at appropriate levels of the hierarchical decomposition of T constructed via restricted recompression. Section 4 provides the implementation details of the restricted recompression technique.

IPM QUERIES *in highly periodic patterns.* As mentioned above, our deterministic sampling strategy applies only to non–highly periodic (NHP) patterns. In section 7, we develop a complementary data structure that is responsible for handling highly periodic (HP) patterns.

For this, we rely on the fact that the structure of all highly periodic fragments can be encoded by *maximal repetitions* (also known as runs) [86, 78, 16]. In particular, we build on the notion of *compatibility* [36]: Two strings are compatible whenever their string periods are cyclically equivalent. If x is periodic, every matching fragment x' can be extended to a run compatible with x. Moreover, due to the assumption $|y| < 2|x|$, if x' is contained within y, then the run must contain the middle position of y. Consequently, we develop a simple new data structure that in $\mathcal{O}(1)$ time lists all runs with a certain minimum length (such as $|x|$) and maximum period (such as $\frac{1}{3}|x|$)

that contain a given position of T. We then use the techniques of [36] to eliminate runs incompatible with x and find the occurrences of x within each compatible run.

IPM Queries *in texts over small alphabets.* The aforementioned techniques allow answering IPM Queries using $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ construction time. In the case of small alphabets, that is, $\sigma = n^{o(1)}$, both complexities can be improved to $\mathcal{O}(n/\log_\sigma n)$. For this, in section 8, we reduce IPM Queries in $T \in [0\,..\,\sigma)^n$ to IPM Queries in a text T' of length $\mathcal{O}(n/\log_\sigma n)$ over an alphabet of size $n^{\Theta(1)}$. We use a τ-synchronizing set Sync, constructed using Proposition 1.11 for appropriate $\tau = \Theta(\log_\sigma n)$, to partition T into blocks, and we encode these blocks as characters of T'. By the density property of Sync, each block has length at most τ or smallest period at most $\frac{1}{3}\tau$. Moreover, the consistency property implies that the block boundaries within x match the block boundaries within any occurrence of x. Consequently, we retrieve the fragments $\Phi(x)$ and $\Phi(y)$ of T', encoding the blocks contained within x and y, respectively; identify the occurrences of $\Phi(x)$ in $\Phi(y)$; and apply LCE Queries (Proposition 1.8) through Lemma 1.14 below to check which of them correspond to occurrences of x in y. A major challenge in implementing this strategy is that $\Phi(y)$ can be much longer than $\Phi(x)$ despite $|y| < 2|x|$. In particular, $\Phi(x)$ can be empty if $|x| = \mathcal{O}(\tau)$ (in that case, we precompute the answers) or if $\operatorname{per}(x) \le \frac{1}{3}\tau$ (in that case, we reuse the techniques for HP patterns). In the remaining cases, we show that it suffices to trim $\Phi(y)$ to a carefully defined fragment of length $\mathcal{O}(|\Phi(x)|)$.

Applications of IPM Queries. Section 9 demonstrates the usage of IPM Queries for answering Prefix-Suffix Queries, Cyclic Equivalence Queries, and Bounded LCP Queries; it also covers applications of all these queries. The common feature of our solutions is that we make a constant number of IPM Queries to list candidates (suffixes for Prefix-Suffix Queries, rotations for Cyclic Equivalence Queries, and previous occurrences for Bounded LCP Queries) and then verify each of them using LCE Queries. In the nonperiodic case, verifying $\mathcal{O}(1)$ candidates does not constitute any significant challenge. Otherwise, we exploit the structure of the output of IPM Queries [21, 96]: The sequence $\mathbf{p} = (p_i)_{i=0}^{k-1}$ of reported starting positions forms a *periodic progression*, meaning that $T[p_0\,..\,p_1) = \cdots = T[p_{k-2}\,..\,p_{k-1})$, and thus, for any position q, the answers to queries $\mathrm{LCE}(p_i, q)$ can be obtained in bulk using just $\mathcal{O}(1)$ LCE queries. Formally, our main auxiliary result reads as follows.

Lemma 1.14. *Consider a text T equipped with a data structure answering* LCE Queries *in $\mathcal{O}(1)$ time. Given a fragment v of T and a collection of fragments $u_i = T[p_i\,..\,r)$ represented with a periodic progression $\mathbf{p} = (p_i)_{i=0}^{k-1}$ and a position $r \ge p_{k-1}$, the following queries can be answered in $\mathcal{O}(1)$ time:*

(a) *Report indices $i \in [0\,..\,k)$ such that u_i matches a prefix of v, represented as a subinterval of $[0\,..\,k)$.*

(b) *Report indices $i \in [0\,..\,k)$ maximizing $\mathrm{lcp}(u_i, v)$, represented as a subinterval of $[0\,..\,k)$.*

2. Strings and periodicity.

We consider *strings* over an *alphabet* Σ, i.e., finite sequences of *characters* from the set Σ. The set of all strings over Σ is denoted by Σ^*, and Σ^+ is the set of nonempty strings over Σ. For an alphabet Σ and an integer $m \in \mathbb{Z}_{\ge 0}$, by Σ^m, we denote the set of length-m strings over Σ. Occasionally, we also work with the family Σ^∞ of *infinite strings* indexed by nonnegative integers.

2.1. Basic notations on strings.

For a string w, by $|w|$, we denote its length and by $w[0], \ldots, w[|w|-1]$ its subsequent characters. The *concatenation* of two strings

u, v is denoted $u \cdot v$ or uv. We identify strings of length 1 with the underlying characters, which lets us write $w = w[0] \cdots w[|w|-1]$. By ε, we denote the empty string. By $alph(w)$, we denote the set $\{w[0], \dots, w[|w|-1]\}$. The string $w^R = w[|w|-1] \cdots w[0]$ is called the *reverse* of w. A string u is called a *substring* of w if $u = w[i] \cdots w[j-1]$ for some $i, j \in [0\,..\,|w|]$ with $i \le j$. In this case, we say that u *occurs* in w at position i, and we denote by $w[i\,..\,j) = w[i\,..\,j-1]$ the *occurrence* of u at position i. By $Occ(u, w)$, we denote the set of starting positions of all occurrences of u in w.

We call $w[i\,..\,j)$ a *fragment* of w; formally, a fragment can be interpreted as a tuple consisting of (a pointer to) the string w and the two endpoints $i, j \in [0\,..\,|w|]$ with $i \le j$. If a fragment $w[i\,..\,j)$ is an occurrence of a string u, then we write $u \cong w[i\,..\,j)$ and say that $w[i\,..\,j)$ *matches* u. Similarly, if $w[i\,..\,j)$ and $w[i'\,..\,j')$ are occurrences of the same string, we denote this by $w[i\,..\,j) \cong w[i'\,..\,j')$, and we say that these fragments match. On the other hand, the equality of fragments $w[i\,..\,j) = w[i'\,..\,j')$ is reserved for occasions when $w[i\,..\,j)$ is the same fragment as $w[i'\,..\,j')$ (i.e., $i = i'$ and $j = j'$).

We assume that a fragment $x = w[i\,..\,j)$ of a string w inherits some notions from the underlying substring: the *length* $|x| = j - i$, the characters $x[i'] = w[i + i']$ for $i' \in [0\,..\,|x|)$, and the *subfragments* $x[i'\,..\,j') = w[i + i'\,..\,i + j')$ for $i', j' \in [0\,..\,|x|)$ with $i' \le j'$. A fragment $w[i\,..\,j)$ also has a natural interpretation as a *range* $[i\,..\,j)$ of positions in w. This lets us consider *disjoint* or *intersecting* (*overlapping*) fragments and define the containment relation ($\subseteq$) on fragments of w. Moreover, for $i, j, k \in [0\,..\,|w|]$ with $i \le j \le k$, the fragments $w[i\,..\,j)$ and $w[j\,..\,k)$ are called *consecutive*, and $w[i\,..\,k) = w[i\,..\,j) \cdot w[j\,..\,k)$ is assumed to be their *concatenation.* If fragments $w[i\,..\,j)$ and $w[i'\,..\,j')$ intersect, we denote their *intersection* $w[\max(i, i')\,..\,\min(j, j'))$ by $w[i\,..\,j) \cap w[i'\,..\,j')$.

A fragment x of w of length $|x| < |w|$ is called a *proper* fragment of w. A fragment $w[i\,..\,j)$ is a *prefix* of w if $i = 0$ and a *suffix* of w if $j = |w|$. We extend the notions of a prefix and a suffix to the underlying substrings. For a string w and an integer $k \in \mathbb{Z}_{\ge 0}$, we denote the concatenation of k copies of w by w^k.

We denote by $\prec$ the natural order on Σ and extend this order in the standard way to the *lexicographic* order on Σ^*.

The notion of concatenation uv extends to $u \in \Sigma^*$ and $v \in \Sigma^\infty$, resulting in $uv \in \Sigma^\infty$. Also, the notion of the longest common prefix of two strings naturally extends to $\Sigma^* \cup \Sigma^\infty$. For a string $w \in \Sigma^+$, we also introduce the infinite power $w^\infty \in \Sigma^\infty$, i.e., the concatenation of infinitely many copies of w.

2.2. Periodic structures in strings. An integer $p \in [1\,..\,|w|]$ is a *period* of a string $w \in \Sigma^+$ if $w[i] = w[i + p]$ holds for all $i \in [0\,..\,|w| - p)$. We call w *periodic* if its smallest period satisfies $\mathrm{per}(w) \le \frac{1}{2}|w|$. A *border* of a string w is a substring of w which occurs both as a prefix and as a suffix of w. Note that p is a period of w if and only if w has a border of length $|w| - p$. Periods of a string w satisfy the following periodicity lemma.

LEMMA 2.1 (periodicity lemma [85, 46]). *Let w be a string with periods p and q. If $p + q - \gcd(p, q) \le |w|$, then $\gcd(p, q)$ is also a period of w.*

For an integer $k \ge 2$, the string w^k is called a *power* of w (with *root* w). A string $u \in \Sigma^+$ is *primitive* if it is not a power, i.e., $u \ne w^k$ for every integer $k \ge 2$ and every root w. For a string u, the shortest string w that satisfies $u = w^k$ for some $k \in \mathbb{Z}_+$ is called the *primitive root* of u. Primitive strings enjoy a synchronizing property, which is a consequence of Lemma 2.1.

LEMMA 2.2 (see [35, Lemma 1.11]). *A nonempty string u is primitive if and only if it occurs exactly twice in u^2 (as a prefix and as a suffix).*

By Lemma 2.2, a string $w \in \Sigma^n$ is primitive if and only if it has exactly n distinct rotations.

Next, we formally state the property that the output of IPM Queries is compact.

Fact 2.3 ([21, 96]). *Let x, y be strings satisfying $|y| < 2|x|$. The set of starting positions of the occurrences of x in y forms a single arithmetic progression.*

Maximal repetitions (runs). A *run* (maximal repetition) [86, 78] in a string w is a periodic fragment $\gamma = w[i\,..\,j]$ that can be extended neither to the left nor to the right without increasing the smallest period $p = \mathrm{per}(\gamma)$, that is, $w[i-1] \neq w[i+p-1]$ and $w[j+1] \neq w[j-p+1]$, provided that the respective positions exist. We assume that runs are stored together with their periods so that $\mathrm{per}(\gamma)$ can be retrieved in constant time. We denote the set of all runs in a string w by $\mathsf{RUNS}(w)$.

Example 2.4. For a string $w = \texttt{baababaababb}$, we have

$$\mathsf{RUNS}(w) = \{w[1\,..\,3), w[6\,..\,8), w[10\,..\,12), w[2\,..\,7), w[7\,..\,11), w[4\,..\,10), w[0\,..\,11)\};$$

see Figure 2. We have three runs with period 1: $w[1\,..\,3) \cong \texttt{aa}$, $w[6\,..\,8) \cong \texttt{aa}$, and $w[10\,..\,12) \cong \texttt{bb}$; two runs with period 2: $w[2\,..\,7) \cong \texttt{ababa}$ and $w[7\,..\,11) \cong \texttt{abab}$; one run with period 3: $w[4\,..\,10) \cong \texttt{abaaba}$; and one run with period 5: $w[0\,..\,11) \cong \texttt{baababaabab}$.

Our results rely on the following asymptotic bounds related to runs.

Proposition 2.5 ([78, 16]). *Given a text T of length n, the set $\mathsf{RUNS}(T)$ of all runs in T (together with their periods) can be computed in $\mathcal{O}(n)$ time. In particular, $|\mathsf{RUNS}(T)| = \mathcal{O}(n)$.*

We say that a run γ *extends* a fragment x if $x \subseteq \gamma$ and $\mathrm{per}(x) = \mathrm{per}(\gamma)$; see Figure 3. Every periodic fragment can be extended to a run with the same period. Moreover, the following easy consequence of Lemma 2.1 implies that this extension is unique. For the fixed text T, we denote the unique run extending a periodic fragment x by $\mathsf{run}(x)$.

Fact 2.6 (see [78, Lemma 1(ii)]). *Let $\gamma \neq \gamma'$ be runs in a string w. If $p = \mathrm{per}(\gamma)$ and $p' = \mathrm{per}(\gamma')$, then $|\gamma \cap \gamma'| < p + p' - \gcd(p, p')$.*

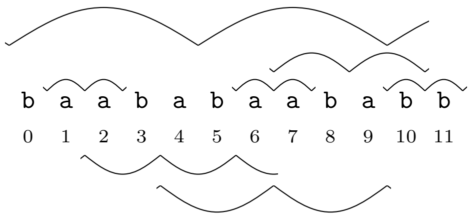


Fig. 2. *Runs in* $w = \texttt{baababaababb}$.

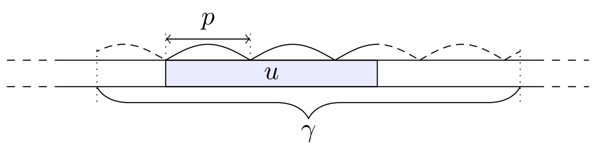


Fig. 3. *A run γ extending a fragment u, that is, $\gamma = \mathsf{run}(u)$, satisfies* $\mathrm{per}(u) = \mathrm{per}(\gamma) = p \leq \frac{1}{2}|u| \leq \frac{1}{2}|\gamma|$.

Proof. For a proof by contradiction, suppose that $|\gamma \cap \gamma'| \geq p + p' - \gcd(p, p')$. By Lemma 2.1, this means that $\gcd(p, p')$ is a period of the intersection $\gamma \cap \gamma'$, which we denote $w[\ell \mathinner{.\,.} r]$. Since $\gamma \neq \gamma'$, exactly one of these runs must contain position $\ell - 1$ or $r + 1$. Due to symmetry, we may assume without loss of generality that γ contains position $\ell - 1$. Observe that positions $\ell + p - 1$ and $\ell + p' - 1$ are located within $\gamma \cap \gamma'$, so $w[\ell + p - 1] = w[\ell + p' - 1]$ because $\gcd(p, p')$ divides $|p - p'|$.

On the other hand, $w[\ell - 1] = w[\ell + p - 1]$ (because γ contains position $\ell - 1$) and $w[\ell - 1] \neq w[\ell + p' - 1]$ (by maximality of γ'). Thus, $w[\ell - 1] = w[\ell + p - 1] = w[\ell + p' - 1] \neq w[\ell - 1]$, which is a contradiction that concludes the proof. ∎

Fragments x satisfying $\mathsf{run}(x) = \gamma$ admit the following elegant characterization.

Observation 2.7. Consider a text T and a run $\gamma \in \mathsf{RUNS}(T)$. A fragment x of T satisfies $\mathsf{run}(x) = \gamma$ if and only if x is contained in γ and $|x| \geq 2\mathrm{per}(\gamma)$.

A string x is called *highly periodic* if $\mathrm{per}(x) \leq \frac{1}{3}|x|$; otherwise, it is called *non–highly periodic.* Highly periodic and non–highly periodic strings are further denoted as HP strings and NHP strings, respectively.

3. Overview of synchronizing sets hierarchy.

Recall that a *τ-synchronizing set* consists of the starting positions of selected length-2τ fragments. It is formally defined as follows.

DEFINITION 3.1 (synchronizing set [65]). *Let T be a string of length n, and let $\tau \in [1 \mathinner{.\,.} \lfloor \frac{n}{2} \rfloor]$. A set $\mathsf{Sync} \subseteq [0 \mathinner{.\,.} n - 2\tau]$ is a* τ-synchronizing set *of T if it satisfies the following conditions:*

Consistency: *For all $i, j \in [0 \mathinner{.\,.} n - 2\tau]$, if $i \in \mathsf{Sync}$ and $T[i \mathinner{.\,.} i + 2\tau) \cong T[j \mathinner{.\,.} j + 2\tau)$, then $j \in \mathsf{Sync}$.*
Density: *For all $i \in [0 \mathinner{.\,.} n - 3\tau + 1]$, we have $[i \mathinner{.\,.} i + \tau) \cap \mathsf{Sync} = \emptyset \iff \mathrm{per}(T[i \mathinner{.\,.} i + 3\tau - 1)) \leq \frac{1}{3}\tau$.*

We say that the elements of a fixed τ-synchronizing set Sync *are* τ-synchronizing positions *and that the fragments $T[s \mathinner{.\,.} s + 2\tau)$ for $s \in \mathsf{Sync}$ are* τ-synchronizing fragments induced by Sync*; the set of τ-synchronizing fragments is denoted* SyncFr.

Example 3.2. Let TM_t be the Thue–Morse word [108] of length 2^t and $\overline{\mathsf{TM}_t}$ be its bitwise negation. For $i \in [0 \mathinner{.\,.} 2^t)$, the character $\mathsf{TM}_t[i]$ is the *pop-count* of i modulo 2 (the parity of ones in the binary representation of i). For $k \in [0 \mathinner{.\,.} t)$, the following construction yields a 2^k-synchronizing set of TM_t:

$$\mathsf{Sync} = \left(Occ(\mathsf{TM}_k, \mathsf{TM}_t) \cup Occ(\overline{\mathsf{TM}_k}, \mathsf{TM}_t)\right) \cap \left[0 \mathinner{.\,.} 2^t - 2^{k+1}\right].$$

We illustrate the case of $t = 5$ and $k = 2$, where $\mathsf{TM}_2 = 0110$. Then $\mathsf{Sync} = \{0, 4, 6, 8, 12, 16, 20, 22, 24\}$ with the 2^2-synchronizing positions underlined:

$$\mathsf{TM}_5 = \underline{0}110\underline{1}0\underline{0}1\underline{1}001\underline{0}110\underline{1}001\underline{0}1\underline{1}0\underline{0}1101001.$$

We say that a fragment is *p-periodic* if its smallest period is at most p; otherwise, the fragment is *p-nonperiodic.* Let us note that if $[i \mathinner{.\,.} i + \tau) \cap \mathsf{Sync} = \emptyset$ holds for a τ-synchronizing set Sync, then the density condition stipulates that $T[i \mathinner{.\,.} i + 3\tau - 1)$ is $\frac{1}{3}\tau$-periodic.

Our construction of a synchronizing sets hierarchy crucially relies on an auxiliary family $(\mathbf{B}_k)_{k=0}^q$ of sets of positions defined below. We denote $\mu = \frac{8}{7}$ and, for $k \in \mathbb{Z}_{\geq 0}$, define

$$\lambda_k = \mu^{\lfloor k/2 \rfloor} \quad \text{and} \quad \alpha_k = \begin{cases} 1 & \text{if } k = 0, \\ \alpha_{k-1} + \lfloor \lambda_{k-1} \rfloor & \text{otherwise.} \end{cases}$$

The numbers λ_k form the sequence $\mu^0, \mu^0, \mu^1, \mu^1, \mu^2, \mu^2, \mu^3, \mu^3, \ldots$. The elements of the sequence α_k are bounded by the elements of the sequence λ_k as follows (see Appendix A for a simple calculation).

Observation 3.3. For every $k \in \mathbb{Z}_{\geq 0}$, we have $\alpha_{k+1} \leq 16\lambda_k$.

In section 4, we prove the following result; the underlying intuition is provided below.

PROPOSITION 3.4. *Given a length-n text T over an alphabet $[0\,..\,n^{\mathcal{O}(1)})$, one can construct in $\mathcal{O}(n)$ time a chain of sets $\emptyset = \mathbf{B}_q \subsetneq \mathbf{B}_{q-1} \subseteq \mathbf{B}_{q-2} \subseteq \cdots \subseteq \mathbf{B}_0 = [1\,..\,n)$ such that each $\mathbf{B}_k$ satisfies the following:*

(a) $|\mathbf{B}_k| \leq 4\,\lambda_k^{-1}\,n$.

(b) *For every $i, j \in [\alpha_k\,..\,n-\alpha_k]$, if $i \in \mathbf{B}_k$ and $T[i-\alpha_k\,..\,i+\alpha_k) \cong T[j-\alpha_k\,..\,j+\alpha_k)$, then $j \in \mathbf{B}_k$.*

(c) *If i, j are consecutive positions in $\mathbf{B}_k \cup \{0, n\}$, then $T[i\,..\,j)$ has length at most $\frac{7}{4}\lambda_k$ or a primitive root of length at most λ_k.*

Restricted recompression. As indicated in section 1.4, we use a modification of the recompression technique [60] to construct a sequence $(\mathsf{F}_0, \mathsf{F}_1, \ldots, \mathsf{F}_q)$ of factorizations of T, where F_0 consists of single characters of T, phrases of F_k are concatenations of phrases of F_{k-1} for $k \in [1\,..\,q]$, and F_q contains one phrase spanning the entire T. The factorizations are represented by the sets $\mathbf{B}_k$ of *phrase boundaries*: the starting positions of all phrases of F_k except for the leftmost one.

The local consistency of recompression is characterized as follows: Whether two subsequent phrases of F_{k-1} are concatenated into the same phrase of F_k depends solely on the *names* of these two phrases (where matching phrases have equal names). Unfortunately, since the phrases can get arbitrarily long, we cannot conclude that the resulting set $\mathbf{B}_k$ of phrase boundaries is locally consistent: For any fixed $k > 1$, whether a given i position belongs to $\mathbf{B}_k$ may depend on a context whose size is not bounded by any function of k. Consequently, in the *restricted* recompression, two subsequent phrases of F_k may be concatenated into the same phrase of F_{k+1} only if their lengths are not "too large."

Our construction (underlying Proposition 3.4) guarantees that every phrase of F_k has length at most $\frac{7}{4}\lambda_k$ or primitive root of length at most λ_k. Moreover, this will also ensure that whether a given i position belongs also to $\mathbf{B}_k$ depends only on its context of length $\alpha_k \leq 16\lambda_{k-1}$.

Synchronizing fragments and synchronizing positions. Since the density condition in Definition 3.1 depends also on the notion of periodicity, our construction needs to capture it as well. For an integer $\tau \in [1\,..\,n]$, we define the set of *τ-runs* in T as

$$\mathsf{RUNS}_\tau(T) = \left\{ \gamma \in \mathsf{RUNS}(T) : |\gamma| \geq \tau \text{ and } \mathrm{per}(\gamma) \leq \frac{1}{3}\tau \right\}.$$

Our synchronizing sets consist of two kinds of positions: those obtained from the sets $\mathbf{B}_k$ and those related to the structure of τ-runs.

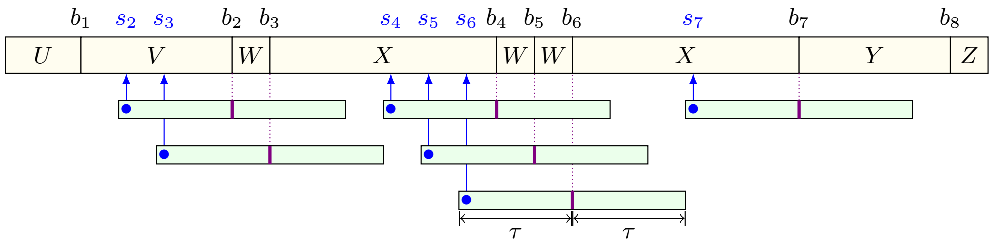


FIG. 4. *Assume* $\mathsf{F}_k = (U, V, W, X, W, W, X, Y, Z)$, *where the symbols* U, V, W, X, Y, Z *are the names of the phrases. The set* $\mathbf{B}_k$ *consists of the starting positions of the phrases. The* τ*-synchronizing set is obtained by going* τ *characters back from the start of each phrase (provided that a length-*2τ *fragment starts at the resulting position). The set* Sync *consists of positions in* T *marked by a blue dot. The figure illustrates the case without highly periodic fragments.*

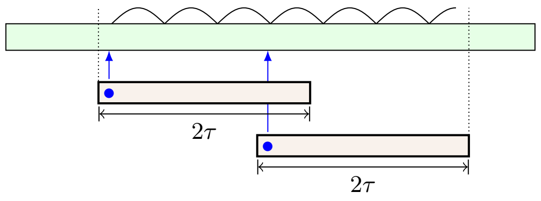


FIG. 5. *Synchronizing positions and synchronizing fragments induced by a* τ*-run. The first (last) positions of such fragments are one position to the left (resp., right) of a* τ*-run. The set* Sync *includes the first positions of synchronizing fragments (blue dots).*

The τ-synchronizing set is based on $\mathbf{B}_k$ at the lowest level k such that $\tau \geq 16\lambda_{k-1} \geq \alpha_k$. Define

$$k(\tau) = \max\{j \in \mathbb{Z}_{\geq 0} : j = 0 \text{ or } 16\lambda_{j-1} \leq \tau\}.$$

The *middle point* of a fragment $T[i\,..\,i+2\ell)$ is defined as $i + \ell$.

CONSTRUCTION 3.5 (sets of synchronizing positions and fragments). *The set* SyncFr *of* τ*-synchronizing fragments consists of all* $\frac{\tau}{3}$*-nonperiodic fragments of length* 2τ *such that*

(a) *their middle point is in* $\boldsymbol{B}_{k(\tau)}$, *or*
(b) *their first position is one position to the left of a* τ*-run, or*
(c) *their last position is one position to the right of a* τ*-run.*

The set Sync *consists of the first positions of fragments in* SyncFr.

Schematic illustrations can be found in Figures 4–6. The procedure constructing the sets $\mathbf{B}_k$ is described and analyzed in section 4. Then, in section 5, we prove that Construction 3.5 indeed yields τ-synchronizing sets and that these sets can be constructed efficiently.

4. Restricted recompression and proof of Proposition 3.4.

In this section, we use a version of recompression [60] to construct a family of sets $\mathbf{B}_k$ satisfying Proposition 3.4; as described above, this set gives the main part of our synchronizing sets hierarchy.

4.1. Definition of restricted recompression.

Given a string $T \in \Sigma^+$, we create a sequence of factorizations $(\mathsf{F}_0, \ldots, \mathsf{F}_q)$, where each factorization F_k decomposes T into m_k *phrases* $T[f_{k,i}\,..\,f_{k,i+1})$ for $i \in [0\,..\,m_k)$. The set of *phrase boundaries*

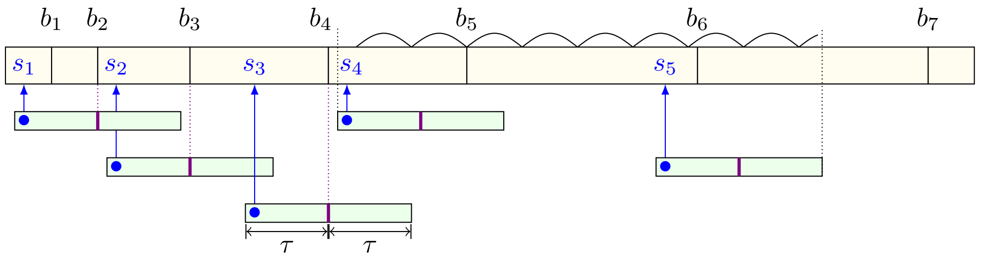


FIG. 6. *Illustration of synchronizing fragments of length 2τ in a general situation: the nonperiodic case and the case of synchronizing fragments generated by τ-runs. The set of τ-synchronizing positions is $\mathsf{Sync} = \{s_1, \dots, s_5\}$, and the set of phrase boundaries is $\mathbf{B}_k = \{b_1, b_2, \dots, b_7\}$.*

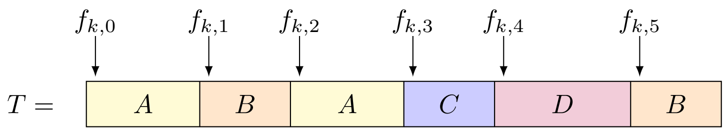


FIG. 7. *The factorization F_k for $m_k = 6$. We have $\mathbf{B}_k = \{f_{k,1}, f_{k,2}, f_{k,3}, f_{k,4}, f_{k,5}\}$ and $T_k = ABACDB$, where $\{A,B,C,D\} \subseteq \mathbf{S}$ are phrase names.*

corresponding to F_k is defined as $\mathbf{B}_k = \{f_{k,1}, \dots, f_{k,m_k-1}\}$; see Figure 7. In particular, if F_k consists of just a single phrase, then $\mathbf{B}_k = \emptyset$.

We identify the factorization F_k with a string $T_k \in \mathbf{S}^+$ of phrase *names* such that $T_k[i] = T_k[j]$ holds if and only if $T[f_{k,i} \,..\, f_{k,i+1}) \cong T[f_{k,j} \,..\, f_{k,j+1})$.

The phrase names belong to an alphabet $\mathbf{S}$ of *symbols* defined as the least fixed point of the following equation:

$$\mathbf{S} = \Sigma \cup (\mathbf{S} \times \mathbf{S}) \cup (\mathbf{S} \times \mathbb{Z}_{\ge 2}).$$

The phrase names can be converted into strings using an *expansion function* $\mathsf{val} : \mathbf{S} \to \Sigma^+$,

$$\mathsf{val}(S) = \begin{cases} S & \text{if } S \in \Sigma, \\ \mathsf{val}(S_1) \cdot \mathsf{val}(S_2) & \text{if } S = (S_1, S_2) \text{ for } S_1, S_2 \in \mathbf{S}, \\ \mathsf{val}(S')^m & \text{if } S = (S', m) \text{ for } S' \in \mathbf{S} \text{ and } m \in \mathbb{Z}_{\ge 2}, \end{cases}$$

that is extended to a *morphism* $\mathsf{val} : \mathbf{S}^* \to \Sigma^*$ with $\mathsf{val}(S_1 \cdots S_s) = \mathsf{val}(S_1) \cdots \mathsf{val}(S_s)$ for $S_1, \dots, S_s \in \mathbf{S}$. Hence, $\mathsf{val}(T_k[j]) \cong T[f_{k,j} \,..\, f_{k,j+1})$ and $\mathsf{val}(T_k) = T$.

Remark 4.1. The characters in $\mathbf{S}$ are rather conceptual; the phrase name can be viewed as a "parse tree" of the phrase (see Figure 8). In an efficient implementation of restricted recompression presented in section 4.2, for each k, we represent the symbols of T_k as small integers.

Next, we describe operations constituting the basic building blocks of restricted recompression. They are modifications of operations used in standard recompression [60].

DEFINITION 4.2 (restricted run-length compression). *Given a string $U \in \mathbf{S}^*$ and a subset $\mathbf{A} \subseteq \mathbf{S}$, we define an operation $\mathsf{\mathbf{RunShrink}}_{\mathbf{A}}(U)$, which returns a string in $\mathbf{S}^*$, as shown in Algorithm 1.*

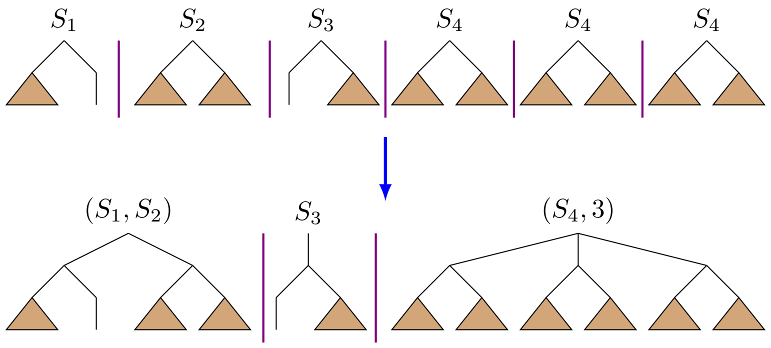


FIG. 8. *Schematic view of one iteration of restricted recompression (with phrase boundaries indicated).*

Algorithm 1: $\mathsf{RunShrink}_{\mathbf{A}}(U)$.

```
foreach maximal unary run $U[i\,..\,i+m) \cong A^m$ in $U$ do
    if $m \geq 2$ and $A \in \mathbf{A}$ then
        replace $U[i\,..\,i+m)$ with the symbol $(A, m) \in \mathbf{S}$;
return $U$;
```

DEFINITION 4.3 (restricted pair compression). *Given a string $U \in \mathbf{S}^*$ and disjoint subsets $\mathbf{L}, \mathbf{L} \subseteq \mathbf{S}$, we define an operation $\mathsf{PairShrink}_{\boldsymbol{L},\boldsymbol{R}}(U)$, which returns a string in $\boldsymbol{S}^*$, as shown in Algorithm 2.*

Algorithm 2: $\mathsf{PairShrink}_{\mathbf{L},\mathbf{R}}(U)$.

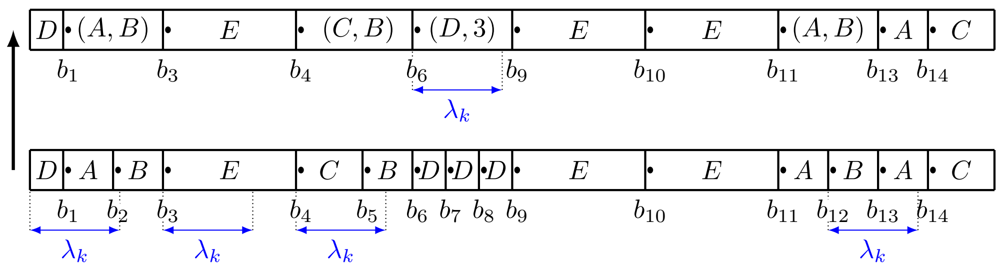
```
foreach fragment $U[i\,..\,i+1]$ in $U$ do
    if $U[i] \in \mathbf{L}$ and $U[i+1] \in \mathbf{L}$ then
        replace $U[i\,..\,i+1]$ with $(U[i], U[i+1]) \in \mathbf{S}$;
return $U$;
```

Let $\mathbf{A}_k := \{S \in \mathbf{S} : |\mathsf{val}(S)| \leq \lambda_k\}$. Given a string $T \in \Sigma^+$, the strings T_k for $k \in \mathbb{Z}_{\geq 0}$ are constructed as shown in Algorithm 3; see also Figure 9.

Algorithm 3: Constructing restricted recompression.

```
$T_0 := T$; $k := 0$;
while $|T_k| > 1$ do
    if $k \bmod 2 = 0$ then
        $T_{k+1} := \mathsf{RunShrink}_{\mathbf{A}_k}(T_k)$;
    else
        $\mathbf{L}_k, \mathbf{R}_k :=$ disjoint subsets of $\mathbf{A}_k$ specified below;
        $T_{k+1} := \mathsf{PairShrink}_{\mathbf{L}_k,\mathbf{R}_k}(T_k)$;
    $k := k+1$;
$q := k$;
```

Construction of $\boldsymbol{L}_k$ and $\boldsymbol{R}_k$. The classification into left symbols $\mathbf{L}_k$ and right symbols $\mathbf{R}_k$ is made similarly as in [59, Lemma 6.2]. We formulate an auxiliary problem and use its folklore deterministic linear-time solution employing the so-called method of conditional expectations [90, section 6.3].



D
• (A, B)
• E
• (C, B)
• (D, 3)
• E
• E
• (A, B)
• A
• C
b_1 b_3 b_4 b_6 b_9 b_{10} b_{11} b_{13} b_{14}
λ_k
D • A • B • E • C • B •D •D •D • E • E • A • B • A • C
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14}
λ_k λ_k λ_k λ_k

FIG. 9. *Detailed illustration of two iterations of computing the restricted recompression, moving from* $T_k = D\,A\,B\,E\,C\,B\,D\,D\,D\,E\,E\,A\,B\,A\,C$ *to* $T_{k+2} = D\,(A,B)\,E\,(C,B)\,(D,3)\,E\,E\,(A,B)\,A\,C$ *for an even* k*. The dots correspond to phrase boundaries. The set of phrase boundaries is changed as follows:* $\mathbf{B}_k = \{b_1, b_2, \dots, b_{14}\} \longrightarrow \mathbf{B}_{k+2} = \{b_1, b_3, b_4, b_6, b_9, b_{10}, b_{11}, b_{13}, b_{14}\}$*. We have* $\mathbf{L}_{k+1} = \{A, C\}$ *and* $\mathbf{R}_{k+1} = \{B, D\}$*. We assume that* $|\mathsf{val}(A)|, \dots, |\mathsf{val}(D)| \le \lambda_k = \lambda_{k+1} < |\mathsf{val}(E)|, |\mathsf{val}((D,3))|$*. In particular,* EE *is not shrunk.*

APPROXIMATE MAXIMUM DIRECTED CUT
Input: A directed multigraph $G = (V, E)$ without self-loops.
Output: A partition $V = L \cup R$ such that at least $\frac{1}{4}|E|$ arcs lead from L to R.

A proof of the following lemma is provided in Appendix A for completeness.

LEMMA 4.4. APPROXIMATE MAXIMUM DIRECTED CUT *problem can be solved in linear time.*

Our construction of $\mathbf{L}_k$ and $\mathbf{R}_k$ works as follows:

- We construct a multigraph with vertices $alph(T_k) \cap \mathbf{A}_k$ and, for $i \in [1\,..\,|T_k|)$, an arc from $T_k[i-1]$ to $T_k[i]$ if both symbols belong to $\mathbf{A}_k$; i.e., both lengths $|\mathsf{val}(T_k[i-1])|$ and $|\mathsf{val}(T_k[i])|$ are at most λ_k.
- The sets $\mathbf{L}_k, \mathbf{R}_k$ are the outputs L, R of the APPROXIMATE MAXIMUM DIRECTED CUT on this multigraph.

The number of arcs from L to R is exactly the number of pairs of merged phrases; the fact that it is bounded from below guarantees that $|\mathbf{B}_k|$ decreases exponentially. This implies, in particular, that $\mathbf{B}_k$ is empty for appropriately large $k = \Theta(\log n)$, so the number of iterations q of the restricted recompression is $\mathcal{O}(\log n)$. The steps using **RunShrink** are not needed to reduce the number of phrases, but they guarantee that there are no self-loops in the constructed multigraphs.

4.2. Proof of Proposition 3.4. Before we proceed with an efficient construction of the family of sets $\mathbf{B}_k$, let us show two basic properties of restricted recompression. We view each application of a shrinking method (**RunShrink** or **PairShrink**) in Algorithm 3 as a decomposition of T_k into fragments, called *blocks*, such that single-character blocks stay intact and longer blocks are *collapsed* into single characters in T_{k+1}. We refer to *block boundaries* as positions of T_k where blocks start. A distinctive feature of recompression (compared to some other locally consistent parsing techniques) is that matching phrases are given equal names. More generally, the following property is shown by induction.

LEMMA 4.5. *For every* $k \in \mathbb{Z}_{\ge 0}$ *and fragments* x, x' *of* T_k*, if the fragments expand to equal strings, that is,* $\mathsf{val}(x) = \mathsf{val}(x')$*, then* x *and* x' *are formed of the same sequences of phrase names, that is,* $x \cong x'$*.*

Proof. We proceed by induction on k. Let x, x' be fragments of T_k satisfying $\mathsf{val}(x) = \mathsf{val}(x')$. If $k = 0$, then $x \cong x'$ holds due to $\mathsf{val}(x) \cong x$ and $\mathsf{val}(x') \cong x'$. Otherwise, let y and y' be the fragments of T_{k-1} obtained from x and x', respectively, by expanding collapsed blocks.

Note that $\mathsf{val}(y) = \mathsf{val}(x) = \mathsf{val}(x') = \mathsf{val}(y')$, so the inductive assumption guarantees $y \cong y'$. Inspecting Definitions 4.2 and 4.3, we can observe that if $T_{k-1}[i-1\,..\,i] \cong T_{k-1}[i'-1\,..\,i']$ for $i, i' \in [1\,..\,|T_{k-1}|)$, then block boundaries at positions i and i' are placed consistently, that is, either at both of them or at neither of them. Consequently, block boundaries within y and y' are placed consistently. Moreover, both y and y' consist of full blocks (since they are collapsed to x and x', resp.). Thus, y and y' are consistently partitioned into blocks. Matching blocks get collapsed to matching symbols in both Definition 4.2 and Definition 4.3, so we derive $x \cong x'$. ∎

In particular, we conclude that $T_k = \mathbf{RunShrink}_{\mathbf{A}_k}(T_k)$ holds for all odd $k \in \mathbb{Z}_{\ge 0}$.

Corollary 4.6. *For every odd $k \in \mathbb{Z}_{\ge 0}$, there is no $j \in [1\,..\,|T_k|)$ such that $T_k[j-1] = T_k[j] \in \mathbf{A}_k$.*

Proof. For a proof by contradiction, suppose that $T_k[j-1] = T_k[j] \in \mathbf{A}_k$ holds for some $j \in [1\,..\,|T_k|)$. By the definition of $\lambda_k = \mu^{\lfloor k/2 \rfloor}$, we have $\mathbf{A}_k = \mathbf{A}_{k-1}$. Let $x = T_{k-1}[i-\ell\,..\,i)$ and $x' = T_{k-1}[i\,..\,i+\ell')$ be blocks of T_{k-1} collapsed to $T_k[j-1]$ and $T_k[j]$, respectively. Due to $\mathsf{val}(x) = \mathsf{val}(T_k[j-1]) = \mathsf{val}(T_k[j]) = \mathsf{val}(x')$, Lemma 4.5 guarantees $x \cong x'$ and, in particular, $\ell = \ell'$. If $\ell = 1$, then $T_{k-1}[i-1] = T_k[j-1] = T_k[j] = T_{k-1}[i] \in \mathbf{A}_k = \mathbf{A}_{k-1}$. Otherwise, $x \cong x' \cong A^\ell$ for some symbol $A \in \mathbf{A}_{k-1}$, which means that $T_{k-1}[i-1] = T_{k-1}[i] = A \in \mathbf{A}_{k-1}$. In either case, $T_{k-1}[i-1] = T_{k-1}[i] \in \mathbf{A}_{k-1}$, which means that $\mathbf{RunShrink}_{\mathbf{A}_{k-1}}(T_{k-1})$ does not place a block boundary at position i in T_{k-1}, a contradiction. ∎

We are ready to prove Proposition 3.4. We repeat its statement for convenience.

Proposition 4.7. *Given a length-n text T over an alphabet $[0\,..\,n^{\mathcal{O}(1)})$, one can construct in $\mathcal{O}(n)$ time a chain of sets $\emptyset = \mathbf{B}_q \subsetneq \mathbf{B}_{q-1} \subseteq \mathbf{B}_{q-2} \subseteq \cdots \subseteq \mathbf{B}_0 = [1\,..\,n)$ such that each $\mathbf{B}_k$ satisfies the following:*

(a) $|\mathbf{B}_k| \le 4\,\lambda_k^{-1}\,n$.

(b) *For every $i, j \in [\alpha_k\,..\,n-\alpha_k]$, if $i \in \mathbf{B}_k$ and $T[i-\alpha_k\,..\,i+\alpha_k) \cong T[j-\alpha_k\,..\,j+\alpha_k)$, then $j \in \mathbf{B}_k$.*

(c) *If i, j are consecutive positions in $\mathbf{B}_k \cup \{0, n\}$, then $T[i\,..\,j)$ has length at most $\frac{7}{4}\lambda_k$ or a primitive root of length at most λ_k.*

Proof. For subsequent integers k, we represent the symbols of T_k via *identifier functions* id_k mapping distinct symbols to distinct identifiers in $[0\,..\,|T_k|)$. Thus, we actually store strings I_k such that $|I_k| = |T_k|$ and $I_k[i] = \mathsf{id}_k(T_k[i])$ for $i \in [0\,..\,|T_k|)$. Moreover, we store arrays mapping identifiers to expansion lengths of the corresponding symbols: $\mathsf{len}_k(I_k[i]) := |\mathsf{val}(T_k[i])|$.

From this representation, it is easy to derive the equality

$$\mathbf{B}_k = \left\{ \sum_{i=0}^{j-1} \mathsf{len}_k(I_k[i]) : j \in [1\,..\,|I_k|) \right\}.$$

In order to construct I_0 and len_0, we sort the characters of T and assign them consecutive positive integer identifiers; the lengths are obviously $\mathsf{len}_0(I_0[i]) = 1$ for all $i \in [0\,..\,|I_0|)$. This step takes $\mathcal{O}(n)$ time due to the assumption $\sigma = n^{\mathcal{O}(1)}$. Moreover, $|\mathbf{B}_0| = n - 1 < 4n = \frac{4n}{\lambda_0}$ holds as claimed.

In order to construct I_{k+1} and len_{k+1} for $k \ge 0$, we process I_k and len_k depending on the parity of k. The algorithm terminates after constructing the first string I_q with $|I_q| = 1$ and even q (then $\mathbf{B}_k = \emptyset$ for $k \ge q$).

Case 1. k is even, that is, $T_{k+1} = \mathsf{\textbf{RunShrink}}_{\mathbf{A}_k}(T_k)$.

We scan I_k from left to right outputting the representations of subsequent symbols of T_{k+1}. The initial representations are elements of $[0\,..\,|T_k|) \cup [0\,..\,|T_k|)^2$; later, they will be given identifiers in $[0\,..\,|T_{k+1}|)$. Each symbol S in T_{k+1} is represented as $(\mathsf{id}_k(S'), m)$ such that $S' \in \mathit{alph}(T_k)$, $m \ge 2$, and $S = (S', m)$ does not occur in T_k or as $\mathsf{id}_k(S)$ otherwise. Suppose that $I_k[j\,..\,|I_k|)$ is yet to be processed. If

$$j = |I_k| - 1, \qquad \mathsf{len}_k(I_k[j]) > \lambda_k, \quad \text{or} \quad I_k[j] \ne I_k[j+1],$$

we output $I_k[j]$ as the next symbol of T_{k+1} and continue processing $I_k[j+1\,..\,|I_k|)$. Otherwise, we determine the maximum integer $m \ge 2$ such that $I_k[j'] = I_k[j]$ for $j' \in [j\,..\,j+m)$, output $(I_k[j], m)$ as the next symbol of T_{k+1}, and continue processing $I_k[j+m\,..\,|I_k|)$. By Lemma 4.5, $(T_k[j], m)$ does not occur in T_k in this case.

Note that $|T_{k+1}| \le |T_k|$. The characters of I_{k+1} are initially represented as elements of $[0\,..\,|T_k|) \cup [0\,..\,|T_k|)^2$, and these representations can be sorted in $\mathcal{O}(|T_k|)$ time so that consecutive integer identifiers id_{k+1} are assigned to symbols of T_{k+1}. We also set

- $\mathsf{len}_{k+1}(\mathsf{id}_{k+1}(S)) = m \cdot \mathsf{len}_k(\mathsf{id}_k(S'))$ if S is represented as $(\mathsf{id}_k(S'), m)$;
- $\mathsf{len}_{k+1}(\mathsf{id}_{k+1}(S)) = \mathsf{len}_k(\mathsf{id}_k(S))$ if S is represented as $\mathsf{id}_k(S)$.

Overall, this algorithm constructs I_{k+1} in $\mathcal{O}(|T_k|) = \mathcal{O}(|\mathbf{B}_k|)$ time.

Case 2. k is odd, that is, $T_{k+1} = \mathsf{\textbf{PairShrink}}_{\mathbf{L}_k, \mathbf{R}_k}(T_k)$.

We first partition $\mathbf{A}_k \cap \mathit{alph}(T_k)$ into $\mathbf{L}_k$ and $\mathbf{R}_k$. Technically, this step results in appropriately marking $\mathsf{id}_k(T_k[i])$ depending on whether $T_k[i] \in \mathbf{L}_k$, $T_k[i] \in \mathbf{R}_k$, or $T_k[i] \notin \mathbf{A}_k$.

For this, we scan the array I_k and construct a directed multigraph G_k with $V(G_k) = \mathit{alph}(I_k)$. For each $i \in [1\,..\,|T_k|)$, we add an arc $I_k[i-1] \to I_k[i]$ provided that

$$\mathsf{len}_k(I_k[i-1]) \le \lambda_k \quad \text{and} \quad \mathsf{len}_k(I_k[i]) \le \lambda_k.$$

By Corollary 4.6, this arc is not a self-loop, so the algorithm of Lemma 4.4 yields in $\mathcal{O}(|\mathbf{B}_k|)$ time a partition $V(G_k) = L \cup R$ with at least $\frac{1}{4}|E(G_k)|$ arcs from L to R. For all symbols S such that $\mathsf{len}_k(\mathsf{id}_k(S)) \le \lambda_k$, we add S to $\mathbf{L}_k$ if $\mathsf{id}_k(S) \in L$ and to $\mathbf{R}_k$ otherwise.

Next, we scan I_k from left to right, outputting representations of subsequent symbols of T_{k+1}. Initially, each symbol S in T_{k+1} is represented as $(\mathsf{id}_k(S_1), \mathsf{id}_k(S_2))$ such that $S_1, S_2 \in \mathit{alph}(T_k)$ and $S = (S_1, S_2)$ does not occur in T_k or as $\mathsf{id}_k(S)$ otherwise. Suppose now that $I_k[j\,..\,|I_k|)$ is yet to be processed. If

$$j = |I_k| - 1, \qquad T_k[j] \notin \mathbf{L}_k, \quad \text{or} \quad T_k[j+1] \notin \mathbf{R}_k,$$

we output $I_k[j]$ as the next symbol of T_{k+1} and continue processing the fragment $I_k[j+1\,..\,|I_k|)$. Otherwise, we output $(I_k[j], I_k[j+1])$ as the next symbol of T_{k+1} and continue processing $I_k[j+2\,..\,|I_k|)$. By Lemma 4.5, $(T_k[j], T_k[j+1])$ does not occur in T_k in this case.

Note that $|T_{k+1}| \le |T_k|$ and that the characters of T_{k+1} are initially represented as elements of $[0\,..\,|T_k|) \cup [0\,..\,|T_k|)^2$. These representations can be sorted in $\mathcal{O}(|T_k|)$ time so that consecutive integer identifiers id_{k+1} can be assigned to symbols of T_{k+1}. We also set

- $\mathsf{len}_{k+1}(\mathsf{id}_{k+1}(S)) = \mathsf{len}_k(\mathsf{id}_k(S_1)) + \mathsf{len}_k(\mathsf{id}_k(S_2))$ if S is represented as $(\mathsf{id}_k(S_1), \mathsf{id}_k(S_2))$;
- $\mathsf{len}_{k+1}(\mathsf{id}_{k+1}(S)) = \mathsf{len}_k(\mathsf{id}_k(S))$ if S is represented as $\mathsf{id}_k(S)$.

Overall, this algorithm constructs I_{k+1} in $\mathcal{O}(|T_k|) = \mathcal{O}(|\mathbf{B}_k|)$ time.

Proof of item (a). If k is even, then $\lambda_k = \lambda_{k+1}$, so $B_{k+1} \subseteq B_k$ implies that

$$|\mathbf{B}_{k+1}| \le |\mathbf{B}_k| \le \frac{4n}{\lambda_k} = \frac{4n}{\lambda_{k+1}}.$$

If k is odd, then we have $|\mathbf{B}_{k+1}| \le |\mathbf{B}_k| - \frac{1}{4}|E(G_k)|$ by construction of the partition $\mathbf{A}_k \cap \mathit{alph}(T_k) = \mathbf{L}_k \cup \mathbf{R}_k$. Observe that, for $i \in [1\,..\,|T_k|)$, an arc $I_k[i-1] \to I_k[i]$ is not added to G_k only if $|\mathsf{val}(T_k[i-1])| > \lambda_k$ or $|\mathsf{val}(T_k[i])| > \lambda_k$. There are at most $\frac{n}{\lambda_k}$ indices $i \in [0\,..\,|T_k|)$ such that $|\mathsf{val}(T_k[i])| > \lambda_k$, and each of them prevents at most two arcs from being added to G_k. Thus, $|E(G_k)| \ge |\mathbf{B}_k| - \frac{2n}{\lambda_k}$, and consequently

$$\begin{aligned}|\mathbf{B}_{k+1}| &\le |\mathbf{B}_k| - \frac{1}{4}|E(G_k)| \le |\mathbf{B}_k| - \frac{1}{4}\left(|\mathbf{B}_k| - \frac{2n}{\lambda_k}\right) = \frac{3}{4}|\mathbf{B}_k| + \frac{n}{2\lambda_k} \le \frac{3n}{\lambda_k} + \frac{n}{2\lambda_k}\\ &= \frac{7n}{2\lambda_k} = \frac{4n}{\lambda_{k+1}}\end{aligned}$$

since $\lambda_{k+1} = \frac{8}{7}\lambda_k$ holds for odd integers $k \in \mathbb{Z}_{\ge 0}$.

Time complexity. The overall running time is

$$\mathcal{O}\left(n + \sum_{k=0}^{q} |\mathbf{B}_k|\right) = \mathcal{O}\left(\sum_{k=0}^{\infty} \frac{4n}{\lambda_k}\right) = \mathcal{O}\left(\sum_{k=0}^{\infty} \left(\frac{7}{8}\right)^{k/2} n\right) = \mathcal{O}(n).$$

Proof of item (b). We proceed by induction on k. The base case of $k = 0$ is trivially satisfied due to $\mathbf{B}_0 = [\alpha_0\,..\,n - \alpha_0]$. Assume now that $k > 0$. For a proof by contradiction, suppose that

$$i \in \mathbf{B}_k \quad \text{and} \quad T[i - \alpha_k\,..\,i + \alpha_k) \cong T[j - \alpha_k\,..\,j + \alpha_k) \quad \text{yet} \quad j \notin \mathbf{B}_k.$$

By $\alpha_k > \alpha_{k-1}$ and the inductive assumption, $i \in \mathbf{B}_k \subseteq \mathbf{B}_{k-1}$ implies that $j \in \mathbf{B}_{k-1}$.

Let us set i', j' so that i and j are the first positions of the phrases induced by $T_{k-1}[i']$ and $T_{k-1}[j']$, respectively, that is,

$$i = f_{k-1,i'} = |\mathsf{val}(T_{k-1}[0\,..\,i'))| \text{ and } j = f_{k-1,j'} = |\mathsf{val}(T_{k-1}[0\,..\,j'))|.$$

Since a block boundary was not placed at $T_{k-1}[j']$, we have (see Definitions 4.2 and 4.3) that $T_{k-1}[j'-1], T_{k-1}[j'] \in \mathbf{A}_{k-1}$. Therefore, the phrases

$$T[j - \ell\,..\,j) \cong \mathsf{val}(T_{k-1}[j'-1]) \text{ and } T[j\,..\,j + r) \cong \mathsf{val}(T_{k-1}[j'])$$

around position j are of length at most $\lfloor \lambda_{k-1} \rfloor$.

Since $\alpha_k = \alpha_{k-1} + \lfloor \lambda_{k-1} \rfloor$, by the inductive assumption, $j - \ell \in \mathbf{B}_{k-1}$ and $j + r \in \mathbf{B}_{k-1}$ imply that $i - \ell \in \mathbf{B}_{k-1}$ and $i + r \in \mathbf{B}_{k-1}$, respectively. Due to Lemma 4.5, this yields

$$T_{k-1}[i'-1] = T_{k-1}[j'-1] \text{ and } T_{k-1}[i'] = T_{k-1}[j'].$$

Consequently, a block boundary was not placed at $T_{k-1}[i']$, which contradicts $i \in \mathbf{B}_k$.

Proof of item (c). We proceed by induction on k. Let S be a symbol in T_k. If $k = 0$, then $|\mathsf{val}(S)| = 1 < \frac{7}{4} \cdot 1 = \frac{7}{4}\lambda_0$. Thus, we may assume that $k > 0$.

If S also occurs in T_{k-1}, then the inductive assumption shows that $\mathsf{val}(S)$ is of length at most $\frac{7}{4}\lambda_{k-1} \le \frac{7}{4}\lambda_k$ or that its primitive root is of length at most $\lambda_{k-1} \le \lambda_k$.

Otherwise, we have two possibilities. If k is odd, then $S = (A,m) \in \mathbf{A}_{k-1} \times \mathbb{Z}_{\ge 2}$, and thus the primitive root of S is of length at most $|\mathsf{val}(A)| \le \lambda_{k-1} = \lambda_k$. If k is even, then $S = (S_1, S_2) \in \mathbf{A}_{k-1} \times \mathbf{A}_{k-1}$, so $|\mathsf{val}(S)| = |\mathsf{val}(S_1)| + |\mathsf{val}(S_2)| \le 2\lambda_{k-1} = \frac{7}{4}\lambda_k$. This completes the proof of the proposition. ◻

5. Details of the synchronizing set hierarchy construction—Proof of Theorem 1.13. In this section, we use Proposition 3.4 and properties of the family $\mathsf{RUNS}_\tau(T)$ of τ-runs to prove that Construction 3.5 yields synchronizing sets of desired size. Moreover, we show how to efficiently build these synchronizing sets. We start with an auxiliary fact.

FACT 5.1. *For every text T and positive integer $\tau \le |T|$, we have $|\mathsf{RUNS}_\tau(T)| < 3|T|/\tau$.*

Proof. By Fact 2.6, distinct τ-runs γ, γ' satisfy $|\gamma \cap \gamma'| < \frac{2}{3}\tau$. Consequently, each τ-run γ contains more than $\frac{1}{3}\tau$ (trailing) positions that are disjoint from all τ-runs starting to the left of γ. ◻

LEMMA 5.2. *Let* Sync *be as in Construction* 3.5 *for a text T of length n and a given $\tau \in [1\,..\,\lfloor \frac{n}{2} \rfloor]$. Then* Sync *is a τ-synchronizing set of size* $|\mathsf{Sync}| < \frac{70n}{\tau}$.

Proof. We prove that Sync satisfies the two conditions of Definition 3.1 and analyze the size of Sync.

Consistency. Suppose that two length-2τ fragments $x = T[i-\tau\,..\,i+\tau)$ and $x' = T[j-\tau\,..\,j+\tau)$ satisfy $x \cong x'$ and $x \in \mathsf{SyncFr}$. We will show that if x satisfies any of the conditions (a)–(c) in Construction 3.5, then x' satisfies the same conditions and $x' \in \mathsf{SyncFr}$.

If x satisfies condition (a), then $i \in \mathbf{B}_{k(\tau)}$, and we need to prove that $j \in \mathbf{B}_{k(\tau)}$. This statement is trivial if $k(\tau) = 0$ due to $\mathbf{B}_0 = [1\,..\,n)$. Otherwise, Observation 3.3 implies that $\alpha_k \le 16\lambda_{k-1}$ for $k = k(\tau)$, and $16\lambda_{k-1} \le \tau$ holds by definition of $k(\tau)$; hence, $x \cong x'$ implies that $T[i-\alpha_k\,..\,i+\alpha_k) \cong T[j-\alpha_k\,..\,j+\alpha_k)$. Thus, $j \in \mathbf{B}_{k(\tau)}$ follows from Proposition 3.4(b). Moreover, if x is $\frac{\tau}{3}$-nonperiodic, then so is $x' \cong x$.

By the next claim, in conditions (b) and (c) of Construction 3.5, we do not need to explicitly mention that the respective synchronizing fragment is $\frac{\tau}{3}$-nonperiodic. The claim follows from Fact 2.6.

CLAIM 5.3. *If the first (last) position of a length-2τ fragment of T is one position to the left (resp., right) of a τ-run, then the fragment is $\frac{\tau}{3}$-nonperiodic.*

If x satisfies condition (b), then

$$\mathrm{per}(T[i-\tau+1\,..\,i]) = \mathrm{per}(T[j-\tau+1\,..\,j]) \le \frac{\tau}{3} < \mathrm{per}(T[i-\tau\,..\,i]) = \mathrm{per}(T[j-\tau\,..\,j]),$$

so there is a τ-run extending the fragment $T[j-\tau+1\,..\,j]$ to the right in T; therefore, x' satisfies condition (b) and $x' \in \mathsf{SyncFr}$ (cf. Claim 5.3).

Condition (c) is symmetric to condition (b); in this case, there is a τ-run extending the fragment $T[j-1\,..\,j+\tau-2]$ to the left in T.

Density. Consider a position $i \in [0\,..\,n-3\tau+1]$. We first show that if $[i\,..\,i+\tau) \cap \mathsf{Sync} = \emptyset$, then $\mathrm{per}(T[i\,..\,i+3\tau-1)) \le \frac{\tau}{3}$.

We start by identifying a τ-run $T[p\,..\,q)$ with $p \le i+\tau-1$ and $i+2\tau$. First, suppose that there exists a position $b \in [i+\tau\,..\,i+2\tau) \cap \mathbf{B}_k$. Then $\mathrm{per}(T[b-\tau\,..\,b+\tau)) \le \frac{\tau}{3}$ since otherwise $T[b-\tau\,..\,b+\tau)$ would have been added to SyncFr by condition (a), and thus

$b-\tau \in [i\,..\,i+\tau)$ would have been added to Sync. Hence, the run $\mathsf{run}(T[b-\tau\,..\,b+\tau))$ starts at position $p \le b-\tau \le i+\tau-1$ and ends at position $q \ge b+\tau \ge i+2\tau$.

Next, suppose that $[i+\tau\,..\,i+2\tau)\cap \mathbf{B}_k = \emptyset$. Then $T[i+\tau-1\,..\,i+2\tau)$ is contained in a single phrase of F_k of length at least $\tau+1$. If $k=0$, then $\tau+1>1$ contradicts the fact that all phrases in F_0 are of length 1. Otherwise, $\tau+1 \ge 16\lambda_{k-1}+1 \ge 14\lambda_k+1 > \frac{7}{4}\lambda_k$, so Proposition 3.4(c) yields

$$\mathrm{per}(T[i+\tau-1\,..\,i+2\tau)) \le \lambda_k < \frac{14}{3}\lambda_k \le \frac{16}{3}\lambda_{k-1} \le \frac{\tau}{3}.$$

Now the τ-run $\mathsf{run}(T[i+\tau-1\,..\,i+2\tau))$ satisfies the desired requirement.

Note that the fragments $T[p-1\,..\,p+2\tau-1)$ and $T(q-2\tau\,..\,q]$ satisfy conditions (b) and (c), respectively (cf. Claim 5.3). Due to $[i\,..\,i+\tau)\cap \mathsf{Sync} = \emptyset$, this implies that $p \le i$ and $q \ge i+3\tau-1$, which means that $\mathrm{per}(T[i\,..\,i+3\tau-1)) \le \frac{\tau}{3}$ holds as claimed.

It remains to show, for all $i \in [0\,..\,n-3\tau+1]$, that if $\mathrm{per}(T[i\,..\,i+3\tau-1)) \le \frac{\tau}{3}$, then $[i\,..\,i+\tau)\cap\mathsf{Sync} = \emptyset$; equivalently, we need to argue that if $[i\,..\,i+\tau)\cap\mathsf{Sync} \ne \emptyset$, then $\mathrm{per}(T[i\,..\,i+3\tau-1)) > \frac{\tau}{3}$. Indeed, if $s \in [i\,..\,i+\tau)\cap\mathsf{Sync}$, then

$$\mathrm{per}(T[i\,..\,i+3\tau-1)) \ge \mathrm{per}(T[s\,..\,s+2\tau)) > \frac{\tau}{3}.$$

Size. Observe that $|\mathsf{Sync}| \le |\mathbf{B}_k| + 2|\mathsf{RUNS}_\tau(T)|$. The second term can be bounded using Fact 5.1. As for $|\mathbf{B}_k|$, we rely on the upper bound of Proposition 3.4(a) and note that $\tau < 16\lambda_k$ holds by the definition of $k = k(\tau)$. Hence,

$$|\mathsf{Sync}| \le |\mathbf{B}_k| + 2|\mathsf{RUNS}_\tau(T)| < \frac{4n}{\lambda_k} + 2\cdot\frac{3n}{\tau} < \frac{64n}{\tau} + \frac{6n}{\tau} = \frac{70n}{\tau}.$$

Before we provide an implementation of Construction 3.5, we need to make sure that the sets $\mathsf{RUNS}_\tau(T)$ can be built efficiently.

LEMMA 5.4. *After $\mathcal{O}(n)$-time preprocessing of a text T of length n, given an integer $\tau \in [1\,..\,n]$, one can construct in $\mathcal{O}(\frac{n}{\tau})$ time the set $\mathsf{RUNS}_\tau(T)$ (with runs ordered by their first positions).*

Proof. For every $k \in \mathbb{Z}_{\ge 0}$, let us define

$$\mathsf{R}_k = \left\{\gamma \in \mathsf{RUNS}(T) : |\gamma| \ge \left(\frac{4}{3}\right)^k \text{ and } \mathrm{per}(\gamma) < \frac{4}{9}\left(\frac{4}{3}\right)^k\right\}.$$

By Fact 2.6, distinct runs $\gamma, \gamma' \in \mathsf{R}_k$ satisfy $|\gamma \cap \gamma'| \le \frac{8}{9}(\frac{4}{3})^k$, and therefore $|\mathsf{R}_k| \le 9\cdot(\frac{3}{4})^k n$. Moreover, note that $\mathsf{RUNS}_\tau(T) \subseteq \mathsf{R}_k$ holds for $k = \lfloor \log_{4/3}\tau \rfloor$ due to

$$\tau \ge \left(\frac{4}{3}\right)^k \quad \text{and} \quad \frac{\tau}{3} = \frac{4}{9}\left(\frac{4}{3}\right)^{(\log_{4/3}\tau)-1} < \frac{4}{9}\left(\frac{4}{3}\right)^k.$$

Preprocessing. In the preprocessing phase, the algorithm computes the values $\lfloor \log_{4/3}\tau \rfloor$ for $\tau \in [1\,..\,n]$ and the sets R_k for $k \in [0\,..\,\lfloor \log_{4/3} n \rfloor]$, with runs ordered by their first positions. In order to construct the sets R_k, the algorithm builds $\mathsf{RUNS}(T)$ using Proposition 2.5 and sorts the runs in $\mathsf{RUNS}(T)$ according to their first positions using bucket sort. Then each $\gamma \in \mathsf{RUNS}(T)$ is added to the appropriate sets R_k, i.e., whenever $k \in [\lfloor \log_{4/3}(\frac{9}{4}\mathrm{per}(\gamma))\rfloor\,..\,\lfloor \log_{4/3}|\gamma| \rfloor]$. The preprocessing time is therefore $\mathcal{O}(n + |\mathsf{RUNS}(T)| + \sum_{k=0}^{\infty}|\mathsf{R}_k|) = \mathcal{O}(n)$.

Queries. At query time, given an integer τ, the algorithm retrieves $k = \lfloor \log_{4/3}\tau \rfloor$ and iterates over $\gamma \in \mathsf{R}_k$, outputting γ whenever $\gamma \in \mathsf{RUNS}_\tau(T)$. The correctness follows from $\mathsf{RUNS}_\tau(T) \subseteq \mathsf{R}_k$, and the query time is $\mathcal{O}(1 + |\mathsf{R}_k|) = \mathcal{O}(\frac{n}{\tau})$ due to $|\mathsf{R}_k| \le 9 \cdot (\frac{3}{4})^k n = 12 \cdot (\frac{3}{4})^{\lfloor \log_{4/3} \tau \rfloor + 1} n < \frac{12n}{\tau}$. ◻

We are ready to prove the main result of this section, that is, $\mathcal{O}(n)$-time construction of a data structure that allows computing a τ-synchronizing set in time $\mathcal{O}(n/\tau)$ for any $\tau \in [1\mathinner{.\,.} \lfloor \frac{n}{2} \rfloor]$. We restate the theorem for convenience.

Theorem 1.13 (construction of synchronizing sets hierarchy). *Given a text T of length n over an alphabet $[0\mathinner{.\,.} n^{\mathcal{O}(1)})$, one can construct in $\mathcal{O}(n)$ time a synchronizing sets hierarchy that, for any $\tau \in [1\mathinner{.\,.} \lfloor \frac{n}{2} \rfloor]$, in $\mathcal{O}(\frac{n}{\tau})$ time returns a τ-synchronizing set* Sync *of T such that $|\mathsf{Sync}| < 70\,\frac{n}{\tau}$ and every τ-synchronizing fragment induced by* Sync *has smallest period larger than $\frac{\tau}{3}$.*

Proof. We separately describe preprocessing and query algorithms.

Preprocessing. The preprocessing phase consists of three steps:

- construct the sets $(\mathbf{B}_k)_{k=0}^q$ using Proposition 3.4;
- perform the preprocessing related to Lemma 5.4;
- compute $k(\tau) = \max\{j \in \mathbb{Z}_{\ge 0} : j = 0 \text{ or } 16\,\lambda_{j-1} \le \tau\}$ for all $\tau \in [1\mathinner{.\,.} \lfloor \frac{n}{2} \rfloor]$.

Query: Constructing τ-synchronizing set. The query algorithm follows Construction 3.5. Synchronizing fragments satisfying conditions (a)–(c) are constructed independently (in left-to-right order), and then the three sorted lists are merged. The construction of all three lists relies on the list $\mathsf{RUNS}_\tau(T)$ of τ-runs obtained from Lemma 5.4. The list is sorted by first positions but also, since no τ-run is contained within another τ-run, by last positions. The τ-synchronizing set is computed in three steps:

- The synchronizing positions satisfying condition (a) are $j - \tau$ for each $j \in \mathbf{B}_{k(\tau)} \cap [\tau\mathinner{.\,.} n-\tau]$ such that $T[j-\tau\mathinner{.\,.} j+\tau)$ is not contained in any τ-run, i.e.,
$$\{T[\ell\mathinner{.\,.} r) \in \mathsf{RUNS}_\tau(T) : \ell \le j-\tau \text{ and } r \ge j+\tau\} = \emptyset.$$
To check this condition, the algorithm simultaneously iterates over positions in $\mathbf{B}_{k(\tau)}$ and the list $\mathsf{RUNS}_\tau(T)$.
- The positions satisfying condition (b) are $\ell - 1$ for every $T[\ell\mathinner{.\,.} r) \in \mathsf{RUNS}_\tau(T)$ with $\ell \in [1\mathinner{.\,.} n-2\tau+1]$. (Recall Claim 5.3.) Thus, they can be generated by iterating over the list $\mathsf{RUNS}_\tau(T)$.
- Finally, the positions satisfying condition (c) are $r - 2\tau + 1$ for every $T[\ell\mathinner{.\,.} r) \in \mathsf{RUNS}_\tau(T)$ with $r \in [2\tau - 1\mathinner{.\,.} n-1]$. Thus, they can be generated by iterating over the list $\mathsf{RUNS}_\tau(T)$.

Overall, constructing Sync costs $\mathcal{O}(1 + |\mathbf{B}_{k(\tau)}| + |\mathsf{RUNS}_\tau(T)|)$ time. As observed in the proof of Lemma 5.2, each of these terms can be bounded by $\mathcal{O}(\frac{n}{\tau})$. Finally, the same lemma guarantees that Sync is a τ-synchronizing set of size $|\mathsf{Sync}| < \frac{70n}{\tau}$. ◻

6. IPM Queries with non–highly periodic patterns.

As discussed in section 1.4, in order to support IPM Queries in the text T, we use a classic idea of pattern matching by deterministic sampling [109] in a novel way. The main trick is to select a consistent family $\mathsf{Samples}$ of *samples.* This allows answering *restricted* IPM Queries, with $x \in \mathsf{Samples}$, using a relatively simple approach in $\mathcal{O}(|\mathsf{Samples}|)$ space; see section 6.2.

In the general case, we first select an arbitrary sample $\hat{x} \in \mathsf{Samples}$ contained in x and search for the occurrences of $\hat{x}$ within y. Then our query algorithm checks which occurrences of $\hat{x}$ can be extended to occurrences of x; see Figure 10. In order to achieve constant query time with this approach, we need to guarantee that $\hat{x}$ has

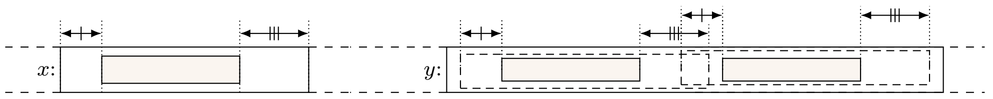


FIG. 10. *The main idea of the query algorithm for fragments x and y. We find the occurrences of $\hat{x} \in \mathsf{Samples}$ contained in y (depicted as gray rectangles). If there is an occurrence x' of x contained in y, then x' can be obtained by extending an occurrence of $\hat{x}$ within y. Hence, x' must be one of the fragments marked with dashed rectangles.*

$\mathcal{O}(1)$ occurrences in y. In general, already x might have $\omega(1)$ occurrences within y. Nevertheless, the following observation can be applied to bound the number of occurrences provided that x does not have a short period. In particular, if $x \in \mathsf{NHP}$ and $|y| < 2|x|$, then x has at most three occurrences within y.

FACT 6.1 (sparsity of occurrences). *If a substring u occurs in a text T at distinct positions i, i', then $|i - i'| \geq \operatorname{per}(u)$.*

Proof. If $i, i + d \in \mathit{Occ}(u, T)$ with $d \in [1\,..\,|u|]$, then $u[j + d] = T[i + j + d] = T[i + j] = u[j]$ for every $j \in [0\,..\,|u| - d)$; i.e., d is a period of u. ◻

6.1. Selection of samples. Let us start with a formal definition based on the discussion above.

DEFINITION 6.2. *A family $\mathsf{Samples}$ of fragments of T is a samples family if it satisfies the following conditions:*

(a) *For all fragments x, x' of T, if $x \cong x'$ and $x \in \mathsf{Samples}$, then $x' \in \mathsf{Samples}$.*

(b) *For every $x \in \mathsf{NHP}$, there exists a fragment $\hat{x} \in \mathsf{Samples}$ contained in x such that $\operatorname{per}(\hat{x}) = \Theta(|x|)$.*

Let us first analyze simple ways to select samples. Arguably, the most naive choice is $\mathsf{Samples} = \mathsf{NHP}$. In this case, the correctness is obvious, with each $x \in \mathsf{NHP}$ being its own sample. However, the number of samples can only be bounded by $\mathcal{O}(n^2)$.

An easy way to dramatically reduce the number of samples is to set $\mathsf{Samples} = \{\hat{x} \in \mathsf{NHP} : \log|\hat{x}| \in \mathbb{Z}\}$ so that $|\mathsf{Samples}| = \mathcal{O}(n \log n)$. Then, as a sample of $x \in \mathsf{NHP}$, we can select an arbitrary fragment $\hat{x} \in \mathsf{NHP}$ of length $2^{\lfloor \log|x| \rfloor}$ contained in x. By the following result, such a fragment always exists.

FACT 6.3. *For every fragment $x \in \mathsf{NHP}$ and length $\ell \in [1\,..\,|x|]$, there is a fragment $\hat{x} \in \mathsf{NHP}$ of length ℓ contained in x.*

Proof. Suppose that $x[0\,..\,\ell) \notin \mathsf{NHP}$. Then there exists a run $\gamma := \mathsf{run}(x[0\,..\,\ell))$ with $\operatorname{per}(\gamma) \leq \frac{1}{3}\ell \leq \frac{1}{3}|x| < \operatorname{per}(x)$. Hence, $x \cap \gamma$ is a proper prefix of x, i.e., $x \cap \gamma = x[0\,..\,i)$ for some $i \in [\ell\,..\,|x|)$. We then define $y = x(i - \ell\,..\,i]$. If $y \notin \mathsf{NHP}$, then there would be another run $\gamma' := \mathsf{run}(y) \neq \gamma$ in x with $\operatorname{per}(\gamma') \leq \frac{1}{3}\ell$. Now $|\gamma \cap \gamma'| \geq |x(i - \ell\,..\,i)| = \ell - 1$ contradicts Fact 2.6. ◻

We aim to select just $\mathcal{O}(n)$ samples. For this, we use synchronizing sets of Theorem 1.13.

LEMMA 6.4. *The family $\mathsf{Samples} = \{T[i\,..\,i] : i \in [0\,..\,n)\} \cup \bigcup_{k=1}^{\lfloor \log n \rfloor} \mathsf{SyncFr}_k$, where SyncFr_k is the set of 2^{k-1}-synchronizing fragments of Theorem 1.13, satisfies Definition 6.2.*

Proof. Let Sync_k denote the 2^{k-1}-synchronizing set containing starting positions of fragments in SyncFr_k. The consistency property (a) follows immediately from the corresponding property of synchronizing sets.

As for the existence of samples (property (b)), let us fix $x \in \mathsf{NHP}$ and choose $k \in [0\,.\,.\,\lfloor \log n \rfloor]$ so that $3 \cdot 2^{k-1} - 1 \le |x| < 3 \cdot 2^k - 1$.

If $k = 0$, then $|x| = 1$, so $x \in \mathsf{Samples}$ can be chosen as its own sample.

Otherwise, let $z \in \mathsf{NHP}$ be a fragment contained in x such that $|z| = 3 \cdot 2^{k-1} - 1$; such a fragment z exists due to Fact 6.3. Observe that

$$\mathrm{per}(z) \ge \frac{1}{3}(|z| + 1) = 2^{k-1} > \frac{1}{3} \cdot 2^{k-1}.$$

Let $z = T[i\,.\,.\,i + |z|)$; by Definition 3.1, $\mathsf{Sync}_k \cap [i\,.\,.\,i + 2^{k-1}) \ne \emptyset$, so we select an arbitrary fragment $\hat{x} := T[s\,.\,.\,s + 2^k) \in \mathsf{SyncFr}_k$ with $s \in [i\,.\,.\,i + 2^{k-1}) \cap \mathsf{Sync}_k$ as the sample of x. Note that $\hat{x}$ is contained in z and that z is contained in x. Moreover,

$$\mathrm{per}(\hat{x}) > \frac{1}{3} \cdot 2^{k-1} = \frac{1}{18} \cdot 3 \cdot 2^k > \frac{1}{18}|x|$$

by the extra condition of Theorem 1.13, and thus $\mathrm{per}(\hat{x}) = \Theta(|x|)$ holds as required. ∎

We conclude with an algorithmic construction based on Lemma 6.4.

Proposition 6.5. *Given a text T of length n, one can in $\mathcal{O}(n)$ time construct a samples family* $\mathsf{Samples}$ *along with a data structure that, given a fragment $x \in \mathsf{NHP}$, in $\mathcal{O}(1)$ time reports a sample $\hat{x} \in \mathsf{Samples}$ contained in x and satisfying $\mathrm{per}(\hat{x}) > \frac{1}{18}|x|$. Moreover, for each $m \in [1\,.\,.\,n]$, we have $|\{\hat{x} \in \mathsf{Samples} : |\hat{x}| \ge m\}| = \mathcal{O}(n/m)$.*

Proof. The construction builds 2^{k-1}-synchronizing sets Sync_k for $k \in [1\,.\,.\,\lfloor \log n \rfloor]$ and the family $\mathsf{Samples}$ as specified in Lemma 6.4. The synchronizing sets are built using Theorem 1.13, which gives $|\mathsf{Sync}_k| = \mathcal{O}(\frac{n}{2^{k-1}})$. Moreover, the construction time is $\mathcal{O}(n + \sum_{k=1}^{\lfloor \log n \rfloor} \frac{n}{2^{k-1}}) = \mathcal{O}(n)$. For every $m \in [1\,.\,.\,n]$, the number of samples of length m or more is $\mathcal{O}(\sum_{k=\lceil \log m \rceil}^{\lfloor \log n \rfloor} \frac{n}{2^{k-1}}) = \mathcal{O}(n/m)$ as claimed.

It remains to efficiently implement assigning samples to fragments $x \in \mathsf{NHP}$, following the approach described in the proof of Lemma 6.4.

The case of $|x| = 1$, when x is its own sample, does not require any infrastructure.

To efficiently find samples for $|x| \in [3 \cdot 2^{k-1} - 1\,.\,.\,3 \cdot 2^k - 1)$ and $k \in [1\,.\,.\,\lfloor \log n \rfloor]$, we store

$$\mathrm{pred}(\mathsf{Sync}_k, i) = \max\{j \le i : j \in \mathsf{Sync}_k\} \text{ and } \mathrm{succ}(\mathsf{Sync}_k, i) = \min\{j > i : j \in \mathsf{Sync}_k\}$$

for each position i divisible by 2^{k-1}. The size and construction time of this component is $\mathcal{O}(\frac{n}{2^k})$, which is $\mathcal{O}(n)$ in total across all values of k.

A query for a sample of $x = T[\ell\,.\,.\,r) \in \mathsf{NHP}$ is answered as follows. As described in the proof of Lemma 6.4, we have $\mathsf{Sync}_k \cap [\ell\,.\,.\,r - 2^k] \ne \emptyset$. Moreover, $r - \ell = |x| \ge 3 \cdot 2^{k-1} - 1$ also yields

$$i := 2^{k-1} \left\lceil \frac{\ell}{2^{k-1}} \right\rceil \in [\ell\,.\,.\,r - 2^k].$$

Consequently, $\mathrm{pred}(\mathsf{Sync}_k, i) \in [\ell\,.\,.\,r - 2^k]$ or $\mathrm{succ}(\mathsf{Sync}_k, i) \in [\ell\,.\,.\,r - 2^k]$, and the underlying fragment can be reported as the sample of x. The query time is constant. ∎

6.2. Implementation of the data structure. As outlined at the beginning of this section, to search for the occurrences of x in y, we first find in y the occurrences of the sample $\hat{x}$ of x. This step is implemented using auxiliary Restricted IPM Queries specified below. Next, we apply LCE Queries (see Proposition 1.8) to check which occurrences of $\hat{x}$ can be extended to occurrences of x; see also Figure 10.

RESTRICTED IPM QUERIES
Input: A text T and a family $\mathsf{Samples}$ of fragments of T.
Queries: Given a fragment $x \in \mathsf{Samples}$ and a fragment y of T, report all fragments $x' \in \mathsf{Samples}$ contained in y and matching x.

Due to the sparsity of occurrences (Fact 6.1), it is relatively easy to implement RESTRICTED IPM QUERIES in $\mathcal{O}(\frac{|y|}{\mathrm{per}(x)})$ time using static deterministic dictionaries.

LEMMA 6.6. *For any family* $\mathsf{Samples}$ *of fragments of a length-n text* T*, there exists a data structure of size* $\mathcal{O}(n + |\mathsf{Samples}|)$ *that answers* RESTRICTED IPM QUERIES *in* $\mathcal{O}(\frac{|y|}{\mathrm{per}(x)})$ *time. It can be constructed in* $\mathcal{O}(n + |\mathsf{Samples}| \log^2 \log |\mathsf{Samples}|)$ *time in general and in* $\mathcal{O}(n + |\mathsf{Samples}|)$ *time if* $|x| = \omega^{\mathcal{O}(1)}$ *for each* $x \in \mathsf{Samples}$*, where* ω *is the machine word size.*

Proof. Given the family $\mathsf{Samples}$, we build an identifier function $\mathsf{id} : \mathsf{Samples} \to \mathbb{Z}$ such that $\mathsf{id}(x) = \mathsf{id}(x')$ if and only if the fragments $x, x' \in \mathsf{Samples}$ match. For this, we order the fragments $x = T[\ell \mathinner{.\,.} r) \in \mathsf{Samples}$ by the length $|x|$ and the lexicographic rank of the suffix $T[\ell \mathinner{.\,.} n)$ among the suffixes of T (this rank is the ℓth entry of the inverse suffix array of T, which can be built in $\mathcal{O}(n)$ time [62]). Matching fragments $x \in \mathsf{Samples}$ appear consecutively in this order, so we use LCE QUERIES (see Proposition 1.8) to determine the boundaries between the equivalence classes.

We store two collections of dictionaries. The first collection allows converting samples to identifiers. The second collection stores nonempty answers to selected RESTRICTED IPM QUERIES.

Dictionaries of identifiers. We store the id function in multiple static dictionaries jointly mapping each fragment $x \in \mathsf{Samples}$ to the identifier $\mathsf{id}(x)$. Specifically, for each position ℓ in T, we store a dictionary $\mathcal{D}(\ell)$ mapping r to $\mathsf{id}(x)$ for every fragment $x = T[\ell \mathinner{.\,.} r) \in \mathsf{Samples}$.

In the general case, we use deterministic dictionaries by Ružić [102], which provide $\mathcal{O}(1)$ query time, take $\mathcal{O}(m)$ space, and have $\mathcal{O}(m \log^2 \log m)$ construction time, where m is the dictionary size. Across all positions ℓ, this brings the overall space consumption to $\mathcal{O}(n + |\mathsf{Samples}|)$ and the overall construction time to $\mathcal{O}(n + |\mathsf{Samples}| \log^2 \log |\mathsf{Samples}|)$. In case of short fragments $|x| = \omega^{\mathcal{O}(1)}$, we use fusion trees [95], which provide $\mathcal{O}(1 + \log_\omega m)$ query time, take $\mathcal{O}(m)$ space, and require $\mathcal{O}(m(1 + \log_\omega m))$ construction time. Since the number of fragments in $\mathsf{Samples}$ starting at any given position ℓ is $\omega^{\mathcal{O}(1)}$ in this case, this brings the overall space consumption and the overall construction time to $\mathcal{O}(n + |\mathsf{Samples}|)$, whereas the query time is $\mathcal{O}(1)$.

Dictionaries of answers to selected queries. For each $k \in [0 \mathinner{.\,.} \lceil \log n \rceil]$, we cover the text T with blocks (fragments) of length $2^{k+1} - 1$ with overlaps of length $2^k - 1$ (the last block can be shorter) and denote the resulting family of blocks by $\mathsf{Y}(k)$. We store the non-empty answers to RESTRICTED IPM QUERIES for $x \in \mathsf{Samples}$ and $y \in \mathsf{Y}(\lceil \log |x| \rceil)$ in multiple static dictionaries. Specifically, for each integer $k \in [0 \mathinner{.\,.} \lceil \log n \rceil]$ and each fragment $y \in \mathsf{Y}(k)$, we store a dictionary $\mathcal{D}'(k, y)$. For each sample $x \in \mathsf{Samples}$ that is contained in y and satisfies $\lceil \log |x| \rceil = k$, the dictionary $\mathcal{D}'(k, y)$ maps the identifier $\mathsf{id}(x)$ to the answer to a RESTRICTED IPM QUERY for x and y. Note that each fragment x is contained in one or two blocks $y \in Y(\lceil \log |x| \rceil)$, so each $x \in \mathsf{Samples}$ appears in $\mathcal{O}(1)$ precomputed answers, and thus the total number of dictionary entries is $\mathcal{O}(|\mathsf{Samples}|)$. If no fragment $x' \in \mathsf{Samples}$ matching $x \in \mathsf{Samples}$ is contained in y, the dictionary $\mathcal{D}'(k, y)$ does not store $\mathsf{id}(x) = \mathsf{id}(x')$.

In the general case, we use deterministic dictionaries by Ružić [102], which provide $\mathcal{O}(1)$ query time, take $\mathcal{O}(n+|\mathsf{Samples}|)$ space in total, and have $\mathcal{O}(n+|\mathsf{Samples}|\log^2\log|\mathsf{Samples}|)$ overall construction time. In case of short patterns, we use fusion trees [95], which provide $\mathcal{O}(1)$ query time, take $\mathcal{O}(n+|\mathsf{Samples}|)$ space in total, and have $\mathcal{O}(n+|\mathsf{Samples}|)$ construction time since each individual dictionary is of size $\omega^{\mathcal{O}(1)}$ in this case (because each $y \in \mathsf{Y}(k)$ contains $\omega^{\mathcal{O}(1)}$ fragments $x \in \mathsf{Samples}$ with $\lceil \log|x|\rceil = k$).

Query algorithm. To answer a query for $x = T[\ell\mathinner{.\,.} r) \in \mathsf{Samples}$ and y, we first compute $k = \lceil \log|x|\rceil$ and $\mathsf{id}(x)$ using $\mathcal{D}(\ell)$. Next, we use simple arithmetic to obtain $\mathcal{O}(\frac{|y|}{|x|})$ blocks $y' \in \mathsf{Y}(k)$ that collectively contain all length-$|x|$ fragments contained in y. We take the union of the precomputed answers for the identifier $\mathsf{id}(x)$ in dictionaries $\mathcal{D}'(k, y')$ to obtain a collection of fragments $x' \in \mathsf{Samples}$ matching x and contained in one of the blocks y'. By Fact 6.1, there are $\mathcal{O}(\frac{|y|}{\mathrm{per}(x)})$ such fragments x', so we can filter and report those contained in y spending $\mathcal{O}(1)$ time on each candidate x'. ◻

Corollary 6.7. *Given the family* $\mathsf{Samples}$ *of samples of a length-n text T constructed through Proposition* 6.5*, one can in $\mathcal{O}(n)$ time construct a data structure that answers* Restricted IPM Queries *in $\mathcal{O}(\frac{|y|}{\mathrm{per}(x)})$ time.*

Proof. We store two instances of the data structure of Lemma 6.6: The first instance, for fragments of length more than ω, contains $\mathcal{O}(\frac{n}{\omega})$ samples, and thus its construction time is $\mathcal{O}(n + \frac{n}{\omega}\log^2\log\frac{n}{\omega}) = \mathcal{O}(n)$. The instance for the remaining fragments, of length at most ω, contains $\mathcal{O}(n)$ samples, and thus its construction time is also $\mathcal{O}(n)$. ◻

We conclude with a full description of the data structure for IPM Queries for NHP patterns.

Proposition 6.8. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ that answers* IPM Queries *in $\mathcal{O}(1)$ time provided that $x \in \mathsf{NHP}$. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Proof. The core of our data structure is the samples family $\mathsf{Samples}$, constructed using Proposition 6.5 along with a component for efficiently selecting a sample, plus the data structure answering Restricted IPM Queries for $\mathsf{Samples}$, constructed using Corollary 6.7. Additionally, we include a data structure for LCE Queries (Proposition 1.8). Each of these ingredients takes $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time to build.

The query algorithm for fragments $x \in \mathsf{NHP}$ and y works as follows. First, we use Proposition 6.5 to obtain in $\mathcal{O}(1)$ time a sample $\hat{x} \in \mathsf{Samples}$ contained in x and satisfying $\mathrm{per}(\hat{x}) = \Theta(|x|)$. Next, we query the component of Corollary 6.7 to find all occurrences of $\hat{x}$ contained in y; this takes $\mathcal{O}(\frac{|y|}{\mathrm{per}(\hat{x})}) = \mathcal{O}(1)$ time. As a result, we obtain a constant number of candidate positions where x may occur in y. We verify them using LCE Queries in $\mathcal{O}(1)$ time each. Recall that the occurrences of x in y form an arithmetic progression (Fact 2.3). ◻

7. IPM Queries with highly periodic patterns.

Our approach to IPM Queries with HP patterns relies on the structure of HP runs in the text. In order to answer a query, we look for runs that may arise as $\mathsf{run}(x')$ for the occurrences x' of x within y. Due to the assumption $|y| < 2|x|$, all these runs contain the middle position of y (the position $T\left[\left\lfloor\frac{\ell+r}{2}\right\rfloor\right]$ if $y = T[\ell\mathinner{.\,.} r)$). All such runs need to have length at least $|x|$ and period at most $\mathrm{per}(x) \le \frac{1}{3}|x|$, which allows us to show that there are $\mathcal{O}(1)$ such runs. In section 7.1, we develop a component listing such runs in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing.

Next, we look for the occurrences of x contained in each candidate run γ. For x to have any occurrence in γ, the periods of x and γ need to be equal; furthermore, the *string periods* of x and γ—that is, the prefixes of the two fragments of length equal to their periods—need to be cyclic rotations of each other. We then say that x and γ are *compatible.* To check this condition, we compare the lexicographically minimal rotations of the string periods, called *Lyndon roots.* We use techniques originating from a paper by Crochemore et al. [36], listed in section 7.2, to check compatibility of substrings of T and list occurrences of an HP pattern in a compatible HP text; in this case, the set of occurrences forms an arithmetic progression.

The query algorithm is described in section 7.3. In brief, for each run γ in y that is compatible with x, we can find all occurrences of x in $y \cap \gamma$ using the toolbox of [36] (recalled in section 7.2). We may obtain $\mathcal{O}(1)$ arithmetic progressions representing the occurrences of x in y, but Fact 2.3 guarantees that they can be merged into a single progression.

7.1. Special HP runs. If x is an HP fragment, then Observation 2.7 yields the following useful characterization of $\mathsf{run}(x)$.

Observation 7.1. Consider a text T and an HP fragment x with $k = \lfloor \log |x| \rfloor$. Then $\gamma = \mathsf{run}(x)$ satisfies the following condition:

$$\text{per}(\gamma) < \frac{1}{3} \cdot 2^{k+1}, \qquad |\gamma| \ge 2^k, \quad \text{and} \quad \gamma \text{ is a } \mathsf{HP} \text{ run.} \tag{7.1}$$

For a positive integer k, we say that a run γ is *k-special* if it satisfies condition (7.1) above. Denote by $\mathsf{SpRuns}_k(i)$ the set of k-special runs covering position i in T. Below, we develop a data structure for efficiently answering the following queries:

Special Run Locating Queries
Given a position i of T and an integer $k \in [0\,..\,\lfloor \log n \rfloor]$, compute $\mathsf{SpRuns}_k(i)$.

We first prove that the answers must be of constant size. Then we develop a component for answering Special Run Locating Queries based on the fact that, even though there are $\Theta(n \log n)$ possible queries, it suffices to precompute answers to $\mathcal{O}(n)$ of them.

Lemma 7.2. $|\mathsf{SpRuns}_k(i)| \le 5$ *for every integer* $k \in [0\,..\,\lfloor \log n \rfloor]$ *and position* i *in* T.

Proof. For a proof by contradiction, suppose that there are at least six such runs $\gamma_j = T[\ell_j\,..\,r_j)$ with $p_j = \text{per}(\gamma_j)$ for $j \in [1\,..\,6]$ and $\ell_1 \le \cdots \le \ell_6$. For each $j \in [1\,..\,6)$, Fact 2.6 yields

$$|\gamma_j \cap \gamma_{j+1}| < p_j + p_{j+1} \le \frac{1}{3}(2^{k+1} + |\gamma_{j+1}|) \le |\gamma_{j+1}|,$$

so γ_{j+1} is not contained in γ_j. Observe also that $|\gamma_j| - 2p_j \ge \frac{1}{3}|\gamma_j| \ge \frac{1}{3} \cdot 2^k$. Consequently,

$$\ell_6 - \ell_1 = \sum_{j=1}^{5}(\ell_{j+1} - \ell_j) = \sum_{j=1}^{5}(|\gamma_j| - |\gamma_j \cap \gamma_{j+1}|) > \sum_{j=1}^{5}(|\gamma_j| - (p_j + p_{j+1}))$$

$$= |\gamma_1| - (p_1 + p_6) + \sum_{j=2}^{5}(|\gamma_j| - 2p_j) \geq |\gamma_1| - 2 \cdot \frac{1}{3} \cdot 2^{k+1} + \sum_{j=2}^{5} \frac{1}{3} \cdot 2^k$$
$$= |\gamma_1| - \frac{2}{3} \cdot 2^{k+1} + \frac{4}{3} \cdot 2^k = |\gamma_1|.$$

We derived $\ell_6 - \ell_1 > |\gamma_1|$, which contradicts $\gamma_1 \cap \gamma_6 \neq \emptyset$. ∎

LEMMA 7.3. *For every text T, there exists a data structure that answers* SPECIAL RUN LOCATING QUERIES *in $\mathcal{O}(1)$ time, takes $\mathcal{O}(n)$ space, and can be constructed in $\mathcal{O}(n)$ time.*

Proof. For every integer $k \in [0\,..\,\lfloor \log n \rfloor]$, the data structure contains the precomputed answers for all positions i divisible by 2^k. Each of these answers takes $\mathcal{O}(1)$ space by Lemma 7.2, so the total size of the data structure is $\mathcal{O}(n)$.

As for the query algorithm, we note that if a k-special run γ covers position i, then, due to $|\gamma| \geq 2^k$, it also covers position $2^k \left\lfloor \frac{i}{2^k} \right\rfloor$ or $2^k \left\lceil \frac{i}{2^k} \right\rceil$. Thus, the query algorithm retrieves the answers for these two positions and, among the obtained k-special runs, reports those covering position i. The query time is constant by Lemma 7.2.

As for the construction algorithm, we enumerate all runs using Proposition 2.5. For each HP run $\gamma = T[\ell\,..\,r)$ and each $k \in [\lfloor \log(3\mathrm{per}(\gamma)) \rfloor\,..\,\lfloor \log|\gamma| \rfloor]$, we append γ to the precomputed answers for all positions $i \in [\ell\,..\,r)$ that are multiples of 2^k. Since there is at least one such position for each considered pair (γ, k), the running time of this process is proportional to the time complexity of the algorithm of Proposition 2.5 plus the total size of the precomputed answers, both of which are $\mathcal{O}(n)$. ∎

7.2. Compatibility of strings and runs. A primitive string $w \in \Sigma^+$ is called a *Lyndon word* [84, 33] if $w \preceq w'$ for every rotation w' of w. Let u be a string with the smallest period $\mathrm{per}(u) = p$. The *Lyndon root* λ of u is the lexicographically smallest rotation of the prefix $u[0\,..\,p)$. We say that two strings are *compatible* if they have the same Lyndon root.

A string u with Lyndon root λ can be uniquely represented as $\lambda'\lambda^k\lambda''$, where λ' is a proper suffix of λ, λ'' is a proper prefix of λ, and $k \in \mathbb{Z}_{\geq 0}$ is a nonnegative integer. The *Lyndon signature* of u is defined as $(|\lambda'|, k, |\lambda''|)$. Note that the Lyndon signature uniquely determines u within its compatibility class. This representation is very convenient for pattern matching if the text is compatible with the pattern.

LEMMA 7.4. *Let x and y be compatible strings. The set of positions where x occurs in y is an arithmetic progression that can be computed in $\mathcal{O}(1)$ time given the Lyndon signatures of x and y.*

Proof. Let λ be the common Lyndon root of x and y, and let their Lyndon signatures be (p, k, s) and (p', k', s'), respectively. Lemma 2.2 (synchronization property) implies that λ occurs in y only at positions i such that $i \equiv p' \pmod{|\lambda|}$. Consequently, x occurs in y only at positions i such that $i \equiv p' - p \pmod{|\lambda|}$. Clearly, x occurs in y at all such positions i within the interval $[0\,..\,|y| - |x|]$. Therefore, it is a matter of simple calculation to compute the arithmetic progression of these positions. ∎

Crochemore et al. [36] showed how to efficiently compute Lyndon signatures of the runs of a given text.

FACT 7.5 ([36]). *There exists an algorithm that, given a text T of length n, in $\mathcal{O}(n)$ time computes Lyndon signatures of all runs in T.*

Finally, we note that the Lyndon signature of a periodic fragment x can be derived from the Lyndon signature of $\mathsf{run}(x)$.

Observation 7.6. Let u be a fragment of a periodic string w such that $|u| \geq 2\text{per}(w)$. Then u is compatible with w. Moreover, given the Lyndon signature of w, one can compute the Lyndon signature of u in $\mathcal{O}(1)$ time.

7.3. Answering queries. Our data structure consists of the set of runs $\mathsf{RUNS}(T)$ (Proposition 2.5), with each run accompanied by its period and Lyndon signature (Fact 7.5), the data structure for LCE Queries (Proposition 1.8), and the data structure of Lemma 7.3 for Special Run Locating Queries. The entire data structure takes $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n)$ time.

As outlined at the beginning of section 7, the query algorithm consists of the following steps:

Algorithm answering IPM Queries with HP patterns
- (A) Compute the Lyndon signature of x.
- (B) Find all $\lfloor \log|x| \rfloor$-special runs γ containing the middle position of y, defined as $T[\lceil \frac{\ell+r}{2} \rceil]$ for $y = T[\ell\,..\,r)$, along with their Lyndon signatures.
- (C) Filter out runs γ incompatible with x.
- (D) For each of the compatible runs γ, compute an arithmetic progression representing the occurrences of x in $y \cap \gamma$.
- (E) Combine the resulting occurrences of x in y into a single arithmetic progression.

Correctness. Clearly, a fragment matching x (and contained in y) starts at each of the reported positions. It remains to prove that no occurrence x' is missed. Since $|y| < 2|x|$, each occurrence x' of x in y contains the middle point of y. Therefore, $\gamma = \mathsf{run}(x')$ is among the runs found in step (B). By Observation 7.6, γ is compatible with x', and, since x and x' match, γ must be compatible with x. Hence, γ is considered in step (D), and the starting position of x' is reported in step (E).

Implementation. In step (A), we use a Special Run Locating Query to list all $\lfloor \log|x| \rfloor$-special runs containing the first position of x, and then we check if any of these runs contains the whole x and has period not exceeding $\frac{1}{3}|x|$. If so, this run is $\mathsf{run}(x)$ by Observation 2.7; otherwise, we raise an error to indicate that $x \in \mathsf{NHP}$. We then use Fact 7.5 and Observation 7.6 to retrieve the Lyndon signature of $\mathsf{run}(x)$ and x, respectively. In step (B), we use another Special Run Locating Query to identify all $\lfloor \log|x| \rfloor$-special runs γ containing the middle position of y. Lemma 7.2 guarantees that we obtain at most five runs γ. In step (C), for each run γ, we use the Lyndon signatures to identify the Lyndon roots of x and γ, represented as fragments of T, and we ask an LCE Query to check if the Lyndon roots match.

For the remaining (compatible) runs γ, we apply Lemma 7.4 to find the occurrences of x in $y \cap \gamma$. There is nothing to do if $|x| > |y \cap \gamma|$. Otherwise, $|y \cap \gamma| \geq |x| \geq 3\text{per}(x) = 3\text{per}(\gamma)$, so Observation 7.6 lets us retrieve the Lyndon signature of $y \cap \gamma$. We are left with at most five arithmetic progressions, one for each compatible run γ. As argued above, their union represents all occurrences of x in y. By Fact 2.3, this set must form a single arithmetic progression. If the progressions are stored by (at most) three elements—the last one and the first two—it is easy to compute the union in constant time.

The above query algorithm also checks if the fragment x is highly periodic. This concludes the proof of the following result.

Proposition 7.7. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ that answers* IPM Queries *in $\mathcal{O}(1)$ time provided that $x \in \mathsf{HP}$ and reports an error whenever $x \in \mathsf{NHP}$. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Combining Proposition 7.7 with Proposition 6.8, we obtain an efficient data structure for IPM Queries over integer alphabets of polynomial size.

Theorem 7.8. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ that answers* IPM Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

In the following section, we show how the data structure can be improved in the case of a small alphabet.

8. IPM Queries in texts over small alphabets. In this section, we assume that $\sigma \leq \sqrt[19]{n}$; otherwise, Theorem 1.2 follows directly from Theorem 7.8. We transform the string T into a string T' of length $\mathcal{O}(n/\tau)$, where $\tau = \lfloor \frac{1}{19}\log_\sigma n \rfloor$. Then each IPM Query in T is reduced to a constant number of IPM Queries in T'. Without loss of generality, we assume that T starts and ends with unique characters to avoid degenerate cases.

A naive idea to construct the string T' would be to partition T into blocks of length τ and interpret each block as an integer with $\tau \cdot \lceil \log \sigma \rceil = \mathcal{O}(\log n)$ bits. Unfortunately, this approach is not helpful: Even if a pattern fragment x has an occurrence in a text fragment y of T, the longest fragment of x consisting of full blocks may have no "aligned" occurrence in the longest fragment of y consisting of full blocks. Therefore, we partition the string into blocks using the elements of a τ-synchronizing set, which can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time for the aforementioned value of τ; see Proposition 1.11.

8.1. Constructing data structure. We use a τ-synchronizing set $\mathsf{Sync} = \{s_0, \dots, s_{m-1}\}$ of T, where $s_0 < \dots < s_{m-1}$, and the corresponding set of synchronizing fragments $\mathsf{SyncFr} = \{f_0, \dots, f_{m-1}\}$, where $f_i = T[s_i \mathinner{.\,.} s_i + 2\tau)$. The aforementioned assumption on T implies that $s_0 \in [0 \mathinner{.\,.} \tau)$ and $s_{m-1} \in (n - 3\tau \mathinner{.\,.} n - 2\tau]$; in particular, $\mathsf{Sync} \neq \emptyset$.

Denote $\Delta_i = s_{i+1} - s_i$ for $i \in [0 \mathinner{.\,.} m-1)$. We construct a length-$(2m-1)$ string T' such that

$$\begin{aligned} T'[2i] &= f_i && \text{for } i \in [0 \mathinner{.\,.} m), \\ T'[2i+1] &= \Delta_i && \text{for } i \in [0 \mathinner{.\,.} m-1). \end{aligned}$$

Every character of T' is either an integer in $[0 \mathinner{.\,.} n]$ or a length-2τ substring of T (which can be interpreted as an integer with $2\tau \lceil \log \sigma \rceil = \mathcal{O}(\log n)$ bits); see Figure 11.

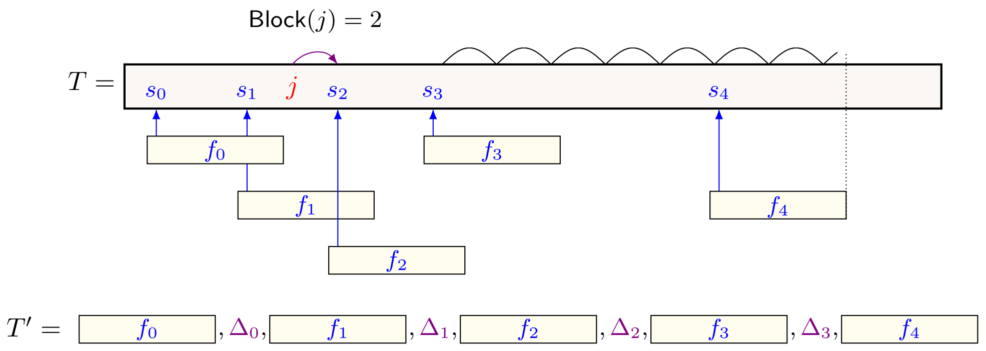
We say that a fragment of T is *regular* if it is of the form $T[s_i \mathinner{.\,.} s_j + 2\tau)$ for $0 \leq i \leq j < m$. For such a fragment z, we denote the fragment $\mathsf{code}(z) = T'[2i \mathinner{.\,.} 2j]$.

Fact 8.1. *If z and z' are regular fragments, then*

$$z \cong z' \iff \mathsf{code}(z) \cong \mathsf{code}(z').$$

Proof. Let $z = T[s_i \mathinner{.\,.} s_j + 2\tau)$ and $z' = T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$.

(*Implication* $\Rightarrow$). Suppose that $T[s_i \mathinner{.\,.} s_j + 2\tau) \cong T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$. We conclude from the consistency property of Sync that $j - i = j' - i'$ and $(\Delta_i, \dots, \Delta_{j-1}) = (\Delta_{i'}, \dots, \Delta_{j'-1})$. This implies that $(f_i, \dots, f_j) \cong (f_{i'}, \dots, f_{j'})$. Hence, $T'[2i \mathinner{.\,.} 2j] \cong T'[2i' \mathinner{.\,.} 2j']$ holds as claimed.



Block$(j) = 2$
$T =$
s_0
s_1
j
s_2
s_3
s_4
f_0
f_1
f_2
f_3
f_4
$T' =$ f_0, Δ_0, f_1, Δ_1, f_2, Δ_2, f_3, Δ_3, f_4

FIG. 11. *A schematic view of the transformation from T to T', assuming* Sync $= \{s_0, \ldots, s_4\}$ *and $\Delta_i = s_i - s_{i-1}$. The* Block *operation returns the index of the nearest synchronizing position if it exists; see below.*

(*Implication* $\Leftarrow$). Suppose that $T'[2i \mathinner{.\,.} 2j] \cong T'[2i' \mathinner{.\,.} 2j']$. Equivalently, we have $(f_i, \ldots, f_j) \cong (f_{i'}, \ldots, f_{j'})$ and $(\Delta_i, \ldots, \Delta_{j-1}) = (\Delta_{i'}, \ldots, \Delta_{j'-1})$. To obtain $T[s_i \mathinner{.\,.} s_j + 2\tau) \cong T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$, we only need to show that if $\Delta_a = \Delta_{a'} > 2\tau$ holds for some a, a' such that $0 \le a - i = a' - i' \le j - i$, then $T[s_a \mathinner{.\,.} s_{a+1}) = T[s_{a'} \mathinner{.\,.} s_{a'+1})$. In this case, by the density property of Sync, we have

$$p := \operatorname{per}(T(s_a \mathinner{.\,.} s_{a+1} + 2\tau - 1)) \le \frac{1}{3}\tau \quad \text{and} \quad p' := \operatorname{per}(T(s_{a'} \mathinner{.\,.} s_{a'+1} + 2\tau - 1)) \le \frac{1}{3}\tau.$$

Since $T[s_a \mathinner{.\,.} s_a + 2\tau) = f_a \cong f_{a'} = T[s_{a'} \mathinner{.\,.} s_{a'} + 2\tau)$, Observation 2.7 implies that $p = p'$ and therefore that $T(s_a \mathinner{.\,.} s_{a+1} + 2\tau - 1) \cong T(s_{a'} \mathinner{.\,.} s_{a'+1} + 2\tau - 1)$. As $f_a \cong f_{a'}$ also yields $T[s_a] = T[s_{a'}]$, this concludes the proof. ∎

Let $s_m = n$ and $s_{-1} = 0$ be sentinels. The sequence $s_{-1}, s_0, \ldots, s_{m-1}, s_m$ of synchronizing positions together with the sentinels partitions $[0 \mathinner{.\,.} n)$ into intervals called *blocks*:

$$[s_{-1} \mathinner{.\,.} s_0),\ [s_0 \mathinner{.\,.} s_1),\ [s_1 \mathinner{.\,.} s_2),\ [s_2 \mathinner{.\,.} s_3), \ldots, [s_{m-1} \mathinner{.\,.} s_m).$$

We define the following operation mapping each position $j \in [1 \mathinner{.\,.} n)$ to the index of the block it belongs to (see Figure 11):

$$\mathsf{Block}(j) = i, \quad \text{where } i \text{ is chosen such that } j \in [s_{i-1} \mathinner{.\,.} s_i).$$

We build a length-n bitmask representing Sync. Then the $\mathsf{Block}(j) = |\{i \in [0 \mathinner{.\,.} m) : s_i \le j\}|$ operation can be viewed as a rank query on this bitmask. These queries can be answered in $\mathcal{O}(1)$ time using a data structure of size $\mathcal{O}(n/\log n)$ that can be constructed in $\mathcal{O}(|\mathsf{Sync}| + n/\log n) = \mathcal{O}(n/\log_\sigma n)$ time [58, 14, 92].

Recall that a run γ is a τ-run if $|\gamma| \ge \tau$ and $\operatorname{per}(\gamma) \le \frac{1}{3}\tau$. We say that a τ-run γ is *long* if $|\gamma| \ge 3\tau - 1$ and use the following proposition for the considered τ.

PROPOSITION 8.2 ([65, section 6.1.2]). *For a positive integer τ, a string $T \in [0 \mathinner{.\,.} \sigma)^n$ contains $\mathcal{O}(n/\tau)$ long τ-runs. Moreover, if $\tau \le \frac{1}{9}\log_\sigma n$, we can compute all long τ-runs in T, compute their Lyndon signatures, and group the long τ-runs by their Lyndon roots in $\mathcal{O}(n/\tau)$ time.*

Let us note that a τ-run is $\lfloor \log \tau \rfloor$-special. Hence, each position in T belongs to at most five τ-runs (Lemma 7.2). Moreover, we can locate long τ-runs using the

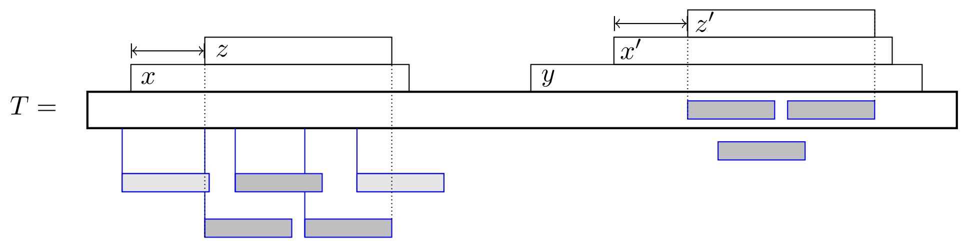


FIG. 12. *We have $z = \mathsf{MaxReg}(x)$ and $z' = \mathsf{MaxReg}(x')$. The arrows correspond to the same distances due to synchronization. Each fragment z' matching z and contained in y determines a single location of a candidate match x'.*

same data structure as in Lemma 7.3 but constructed only for $k = \lfloor \log \tau \rfloor$ (and using Proposition 8.2 to compute all long τ-tuns).

LEMMA 8.3. *For every text T, there exists a data structure that can compute all long τ-runs containing a given position in $\mathcal{O}(1)$ time, takes $\mathcal{O}(n/\tau)$ space, and can be constructed in $\mathcal{O}(n/\tau)$ time.*

We compute T^R in $\mathcal{O}(n/\log_\sigma n)$ time with tabulation, reversing one half-word at a time (cf. [17]). We also perform preprocessing for LCE QUERIES (Proposition 1.8) on T and on T^R.

Finally, for every pair (x, y) of strings such that $|x| < 8\tau$ and $|y| < 10\tau$, we precompute the set of occurrences of x in y, represented as an arithmetic progression. The total number of such pairs (x, y) is $\mathcal{O}(\sigma^{18\tau}) = \mathcal{O}(n^{18/19})$, and for each such pair, the output can be computed in $\mathcal{O}(\tau^{\mathcal{O}(1)}) = n^{o(1)}$ time, so this preprocessing can be performed in $\mathcal{O}(n/\tau)$ time.

8.2. Answering queries. We denote by $\mathsf{MaxReg}(u)$ the longest regular fragment contained in the fragment u of T and denote $\Phi(u) = \mathsf{code}(\mathsf{MaxReg}(u))$. Note that, for any two matching fragments $u \cong u'$, we have $\mathsf{MaxReg}(u) \cong \mathsf{MaxReg}(u')$ (by consistency of Sync) and $\Phi(u) \cong \Phi(u')$ (by Fact 8.1); see Figure 12.

Observation 8.4. After $\mathcal{O}(n/\tau)$-time preprocessing, given fragments x, y of T, we can compute the fragments $\mathsf{MaxReg}(x)$, $\mathsf{MaxReg}(y)$ in T and their codes $\Phi(x)$, $\Phi(y)$ in T' in $\mathcal{O}(1)$ time.

First, we consider the case when $|x| \geq 8\tau$ and $\mathsf{MaxReg}(x) \neq \varepsilon$; the remaining corner cases will be addressed later. In IPM QUERIES, we assume that the length of y is proportional to the length of x. Here, we will make a stronger assumption that $|x| \leq |y| \leq \frac{5}{4}|x|$. However, this assumption does not imply immediately that $|\Phi(y)|$ is proportional to $|\Phi(x)|$. The latter condition is needed to apply IPM QUERIES to fragments $\Phi(x)$ and $\Phi(y)$ because $|\Phi(y)|$ could be too large compared with $|\Phi(x)|$.

Denote $y = T[\ell \mathinner{.\,.} r]$, and let $\mathsf{Mid}(y) = \left\lfloor \frac{\ell + r}{2} \right\rfloor$ be the middle position of y. Our approach, in this case, is to restrict the search to an appropriate fragment of length at most $2|\Phi(x)| + 1$,

$$\Phi'(y) = T'[\,2i - |\Phi(x)| - 1 \mathinner{.\,.} 2i + |\Phi(x)| - 1\,] \cap \Phi(y), \text{ where } i = \mathsf{Block}(\mathsf{Mid}(y)),$$

equal to an approximately "middle" part of $\Phi(y)$; see Figure 13.

FACT 8.5. *Let x and y be fragments of T such that $8\tau \leq |x| \leq |y| \leq \frac{5}{4}|x|$, and let x' be a fragment matching x and contained in y. If $\mathsf{MaxReg}(x) \neq \varepsilon$, then $\Phi(x')$ contains $T'[2i - 2]$ or $T'[2i]$, where $i = \mathsf{Block}(\mathsf{Mid}(y))$.*

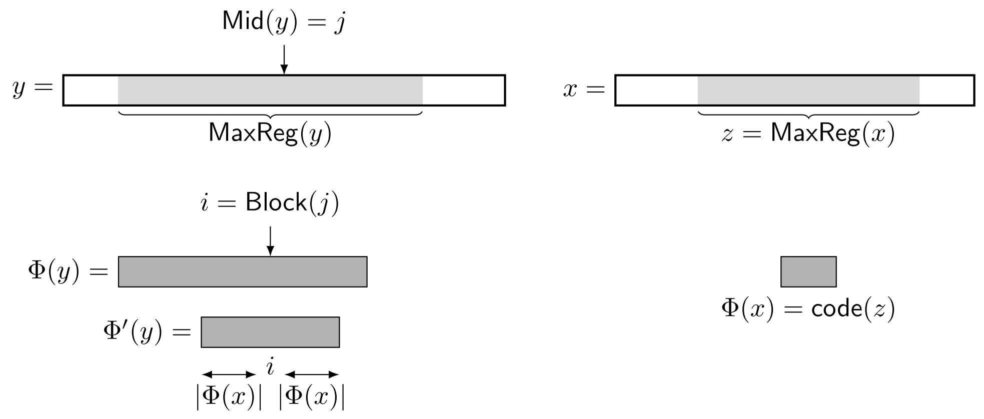


FIG. 13. *Instead of searching for $\Phi(x)$ within $\Phi(y)$, we only search within a fragment $\Phi'(y)$ of length at most $2|\Phi(x)|+1$. Recall that each character of $\Phi(x)$ and $\Phi(y)$ fits in a single machine word.*

Proof. Recall that $\mathsf{MaxReg}(x) \cong \mathsf{MaxReg}(x')$, so $\mathsf{MaxReg}(x') \neq \varepsilon$. Let $z' = \mathsf{MaxReg}(x') = T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$ so that $\Phi(x') = T[2i' \mathinner{.\,.} 2j']$. It suffices to prove that $\{i-1, i\} \cap [i' \mathinner{.\,.} j'] \neq \emptyset$.

- If $\mathsf{Mid}(y) \in (s_{i'-1} \mathinner{.\,.} s_{j'+1})$, then $s_i \in [s_{i'} \mathinner{.\,.} s_{j'+1}]$, so $i \in [i' \mathinner{.\,.} j'+1]$ and $\{i-1, i\} \cap [i' \mathinner{.\,.} j'] \neq \emptyset$.
- If $\mathsf{Mid}(y) \le s_{i'-1}$, then, since both $T[\mathsf{Mid}(y)]$ and $z' = T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$ are contained in x', we conclude that $T[s_{i'-1} \mathinner{.\,.} s_{j'} + 2\tau)$ is contained in x', contradicting the definition of $\mathsf{MaxReg}(x')$.
- Finally, if $\mathsf{Mid}(y) \ge s_{j'+1}$, then we claim that $T[s_i \mathinner{.\,.} s_{j'+1} + 2\tau)$ is contained in x', contradicting the definition of $\mathsf{MaxReg}(x')$. Indeed, $\frac{1}{2}|y| + 2\tau \le \frac{5}{8}|x| + \frac{1}{4}|x| < |x|$, so x' contains both $T[s_{i'} \mathinner{.\,.} s_{j'} + 2\tau)$ and $T[\mathsf{Mid}(y) \mathinner{.\,.} \mathsf{Mid}(y) + 2\tau)$ and thus also $T[s_{i'} \mathinner{.\,.} s_{j'+1} + 2\tau)$. ∎

In the query algorithm below, we assume without loss of generality that $|x| \le |y| \le \frac{5}{4}|x|$. (To handle $|y| \le 2|x|$, we combine the answers of up to four queries.)

Algorithm answering IPM QUERIES over small alphabets

(A) If $|x| < 8\tau$, return a precomputed answer.
(B) If $\mathsf{MaxReg}(x) = \varepsilon$, apply the query algorithm for HP patterns of section 7.3.
(C) Apply IPM QUERIES for pattern $\Phi(x)$ and text $\Phi'(y)$ (Theorem 7.8), obtaining at most two arithmetic progressions of occurrences.
(D) Apply Lemma 1.14(b) for T and T^R to test which of these occurrences extend to occurrences of x.

Correctness. In step (B), if $\mathsf{MaxReg}(x) = \varepsilon$, then the density of Sync implies that $\mathrm{per}(x) \le \frac{1}{3}\tau$, so indeed x is an HP pattern.

In step (C), we have $|\Phi'(y)| \le 2|\Phi(x)| + 1$, so Fact 2.3 implies that there are up to two arithmetic progressions.

In step (D), we only care about occurrences of $\Phi(x)$ starting at even positions of $\Phi'(y)$. Each arithmetic progression from step (C) forms a periodic progression $\mathbf{p} = (p_i)_{i=0}^{k-1}$ in T. This is because, by Fact 8.1, subsequent occurrences of $z = \mathsf{MaxReg}(x)$ start at all these positions. Let $x = wzw'$ and $y = T[\ell \mathinner{.\,.} r)$. We apply Lemma 1.14(b) in T to sequence $\mathbf{p}$, position r, and fragment zw' to check which positions p_i contain occurrences of zw'. Then we apply Lemma 1.14(b) in T^R to sequence $\mathbf{p}' = (n - p_{k-1-i})_{i=0}^{k-1}$, position $n - \ell$, and fragment w^R to check which positions $p_i - 1$ in T are endpoints of occurrences of w. In each case, the lemma returns an integer interval of indices. The intersection of the two intervals can be transformed into an arithmetic progression of positions. The two resulting arithmetic progressions can be joined together to one progression by Fact 2.3.

Implementation. Recall that T is given in a packed representation. In step (A), this lets us retrieve any fragment of length at most 10τ, encoded in a single machine word, in $\mathcal{O}(1)$ time. Then we can use the precomputed answer.

In step (B), the preprocessing of the query algorithm of section 7.3 takes only $\mathcal{O}(n/\tau)$ time and space as we use Lemma 8.3 to find long τ-runs and the LCE queries of Proposition 1.8.

In step (C), we have $|T'| = \mathcal{O}(n/\tau)$, and T' can be extracted from the packed representation of T in $\mathcal{O}(n/\tau)$ time. The preprocessing of IPM Queries of Theorem 7.8 on T' takes $\mathcal{O}(n/\tau)$ time and space.

For step (D), we compute T^R in $\mathcal{O}(n/\log_\sigma n)$ time and use LCE queries of Proposition 1.8.

This concludes the description of our data structure for IPM Queries.

Theorem 1.2 (main result). *For every text $T \in [0\..\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* IPM Queries *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

9. Applications of IPM and LCE queries.

We present some applications of our data structure for Internal Pattern Matching Queries. This includes answering Period Queries (section 9.2), Cyclic Equivalence Queries (section 9.3), and variants of LZ Substring Compression Queries (section 9.4). Before that, we prove Lemma 1.14, which is useful in all our applications, as discussed in section 1.4.

9.1. Proof of Lemma 1.14.

Lemma 1.14 is restated and proved below. Before that, we present useful notations and facts.

Let us recall that a sequence $\mathbf{p}$ of positions $p_0 < p_1 < \cdots < p_{k-1}$ in a string w is a *periodic progression* of length k (in w) if $w[p_0\..p_1) \cong \cdots \cong w[p_{k-2}\..p_{k-1})$. If $k \geq 2$, we call the string $v \cong w[p_i\..p_{i+1})$ the *(string) period* of $\mathbf{p}$, while its length $p_{i+1} - p_i$ is the *difference* of $\mathbf{p}$. Periodic progressions $\mathbf{p}, \mathbf{p}'$ are called *nonoverlapping* if the last term of $\mathbf{p}$ is smaller than the first term of $\mathbf{p}'$ or vice versa: the last term of $\mathbf{p}'$ is smaller than the first term of $\mathbf{p}$. Every periodic progression is an arithmetic progression and, consequently, can be represented by three integers, e.g., the terms p_0, p_1, and p_{k-1} (with p_1 omitted if $k=1$, i.e., if $p_{k-1}=p_0$).

All our applications of IPM Queries rely on the structure of the values $\mathrm{LCE}(p_i, q)$ for a periodic progression $(p_i)_{i=0}^{k-1}$. In Lemma 1.14 below, we give a combinatorial characterization of this structure (in a slightly more general form) amended with its immediate algorithmic applications. Let us start with a simple combinatorial result.

Fact 9.1. *For strings $u, v \in \Sigma^*$ and $\rho \in \Sigma^+$, let $d_u = \mathrm{lcp}(\rho^\infty, u)$ and $d_v = \mathrm{lcp}(\rho^\infty, v)$.*

(a) *If $d_u \neq d_v$, then $\mathrm{lcp}(u,v) = \min(d_u, d_v)$.*

(b) *If $d_u = d_v$, then $\mathrm{lcp}(u,v) \geq d_u = d_v$.*

Proof. Let $d = \min(d_u, d_v)$. Note that $u[0\..d) \cong (\rho^\infty)[0\..d) \cong v[0\..d)$, so $\mathrm{lcp}(u,v) \geq d$. If $d = d_u < d_v$, then $v[d] = (\rho^\infty)[d] \neq u[d]$, so $\mathrm{lcp}(u,v) = d$. The case of $d = d_v < d_u$ is symmetric. ∎

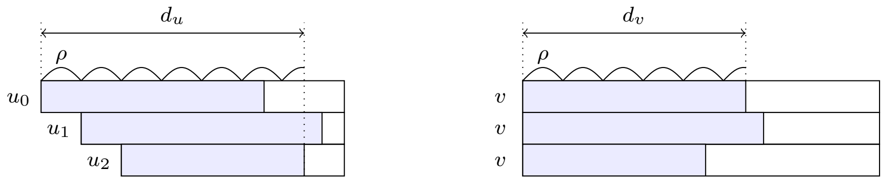


FIG. 14. *An illustration of notions used in Lemma* 1.14. *Shaded rectangles represent the common prefixes of u_i and v. In this case, $\frac{d_u-d_v}{|\rho|}=1$.*

FACT 9.2 (applications of LCE QUERIES). *Assume that we have access to a text T equipped with a data structure answering* LCE QUERIES *in $\mathcal{O}(1)$ time. Given fragments x,y of T, the values $\operatorname{lcp}(x,y)$ and $\operatorname{lcp}(x^\infty,y)$ can be computed in $\mathcal{O}(1)$ time.*

Proof. If $x = T[i_x\,..\,j_x)$ and $y = T[i_y\,..\,j_y)$, then $\operatorname{lcp}(x,y)$ can be computed in constant time due to $\operatorname{lcp}(x,y)=\min(\operatorname{LCE}(i_x,i_y),|x|,|y|)$.

If $\operatorname{lcp}(x,y) < |x|$, i.e., x is not a prefix of y, then $\operatorname{lcp}(x^\infty,y) = \operatorname{lcp}(x,y)$. Otherwise, consider a fragment $y' = T[i_y + |x|\,..\,j_y)$. A simple inductive proof shows that $\operatorname{lcp}(x^\infty,y)=|x|+\operatorname{lcp}(x^\infty,y')=|x|+\operatorname{lcp}(y,y')$. In either case, $\operatorname{lcp}(x^\infty,y)$ can be computed in constant time. ∎

Although our applications use Fact 9.1 for two different purposes, the overall scheme is the same each time. Consequently, we group two application-specific queries in a single algorithmic lemma; see Figure 14.

LEMMA 9.3. *Consider a text T equipped with a data structure answering* LCE QUERIES *in $\mathcal{O}(1)$ time. Given a fragment v of T and a collection of fragments $u_i = T[p_i\,..\,r)$ represented with a periodic progression $\mathbf{p}=(p_i)_{i=0}^{k-1}$ and a position $r\ge p_{k-1}$, the following queries can be answered in $\mathcal{O}(1)$ time:*

(a) *Report indices $i\in[0\,..\,k)$ such that u_i matches a prefix of v, represented as a subinterval of $[0\,..\,k)$.*

(b) *Report indices $i\in[0\,..\,k)$ maximizing $\operatorname{lcp}(u_i,v)$, represented as a subinterval of $[0\,..\,k)$.*

Proof. There is nothing to do for $k = 0$. For $k = 1$, Fact 9.2 lets us check if $\operatorname{lcp}(u_0,v)=|u_0|$, that is, whether u_0 matches a prefix of v. Moreover, $i=0$ maximizes $\operatorname{lcp}(u_i,v)$.

Henceforth, we shall assume that $k\ge 2$. In this case, we retrieve an occurrence $T[p_0\,..\,p_1)$ of the string period ρ of $\mathbf{p}$ and apply Fact 9.2 to determine $d_u=\operatorname{lcp}(\rho^\infty,u_0)$ and $d_v=\operatorname{lcp}(\rho^\infty,v)$. We also compute $i_t=\frac{d_u-d_v}{|\rho|}$.

Let us observe that for $i\in[0\,..\,k)$, $u_0=\rho^i u_i$, so $\operatorname{lcp}(\rho^\infty,u_i)=\operatorname{lcp}(\rho^\infty,u_0)-i|\rho| = d_u-i|\rho|$. If $i_t\in[0\,..\,k)$, then $\operatorname{lcp}(\rho^\infty,u_{i_t})=d_v$. Hence, by Fact 9.1, we have $\operatorname{lcp}(u_i,v)=d_v$ for $i<i_t$, $\operatorname{lcp}(u_i,v)\ge d_v$ for $i=i_t$ (if $i_t\in[0\,..\,k)$), and $\operatorname{lcp}(u_i,v)=d_u-i|\rho|<d_v$ for $i>i_t$.

(a) We shall report $i\in[0\,..\,k)$ such that $\operatorname{lcp}(u_i,v)=|u_i|$. For $i<i_t$, we have $\operatorname{lcp}(u_i,v)=d_v<d_u-i|\rho|\le|u_0|-i|\rho|=|u_i|$, so these indices are never reported. If $i_t\in[0\,..\,k)$, we compute $\operatorname{lcp}(u_{i_t},v)$ using Fact 9.2, and this index may need to be reported. For $i>i_t$, we have $\operatorname{lcp}(u_i,v)=d_u-i|\rho|$ and $|u_i|=|u_0|-i|\rho|$, so we report either all these indices (if $d_u=|u_0|$) or none of them (otherwise).

(b) If $i_t\in[0\,..\,k)$, we check whether $\operatorname{lcp}(u_{i_t},v)>d_v$ using Fact 9.2. If so, we report i_t as the only index maximizing $\operatorname{lcp}(u_i,v)$ because $\operatorname{lcp}(u_i,v)\le d_v$ holds unless $i=i_t$. Otherwise, the maximum of $\operatorname{lcp}(u_i,v)$ is d_v, attained for all $i\in[0\,..\,k)$ such that $i\le i_t$ (if $i_t\ge 0$), or d_u, attained for $i=0$ (if $i_t\le 0$). ∎

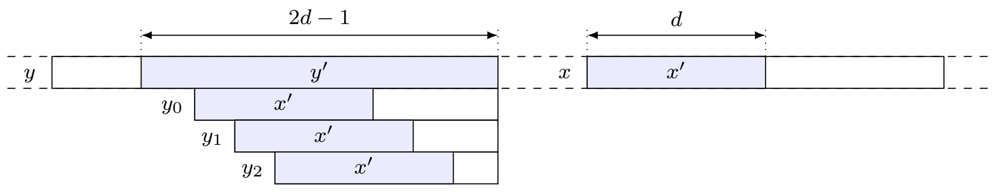


FIG. 15. *The notions used in the algorithms answering* PREFIX-SUFFIX QUERIES *and* BOUNDED LCP QUERIES *(for the latter, see section* 9.4*).*

9.2. Prefix-suffix queries and their applications. In this section, we show the solutions for PREFIX-SUFFIX QUERIES, PERIOD QUERIES, and PERIODIC EXTENSION QUERIES using IPM QUERIES. We start with PREFIX-SUFFIX QUERIES. Let us recall that in this problem, we are given fragments x and y of T and a positive integer d and are to report all suffixes of y of length in $[d\mathinner{.\,.}2d)$ that also occur as prefixes of x. The output is represented as an arithmetic progression of the lengths of suffixes.

Assume that $|x|,|y| \ge d$; otherwise, there are no suffixes to be reported. Let x' be the prefix of x of length d and y' be the suffix of y of length $\min(2d-1,|y|)$. Suppose that a suffix z of y matches a prefix of x. If $|z| \ge d$, then z must start with a fragment matching x'. Moreover, if $|z| \le 2d-1$, then z is a suffix of y', so this yields an occurrence of x' in y'. We find all such occurrences with a single IPM QUERY and then use Lemma 1.14 to find out which of them can be extended to the sought suffixes z of y.

By Fact 2.3, the starting positions of the occurrences of x' in y' form a periodic progression in T. Let y_i be the suffix of y starting with the ith occurrence of x'; see Figure 15. We need to check which of the fragments y_i occur as prefixes of x. This is possible using Lemma 1.14(a), which lets us find all indices i such that y_i is a prefix of x. The result is an integer interval of indices, which can be transformed into an arithmetic progression of lengths $|y_i|$. Consequently, the data structure of Theorem 1.2 (which already contains the component of Proposition 1.8 for LCE QUERIES) can answer PREFIX-SUFFIX QUERIES in $\mathcal{O}(1)$ time. Hence, we obtain the following results.

THEOREM 1.4. *For every text $T \in [0\mathinner{.\,.}\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* PREFIX-SUFFIX QUERIES *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

THEOREM 1.3. *For every text $T \in [0\mathinner{.\,.}\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* PERIOD QUERIES *in $\mathcal{O}(\log|x|)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Proof. PERIOD QUERIES can be answered using PREFIX-SUFFIX QUERIES for $y = x$. To compute all periods of x, we use PREFIX-SUFFIX QUERIES to find all borders of x of length within $[2^k\mathinner{.\,.}2^{k+1})$ for each $k \in [0\mathinner{.\,.}\lfloor\log|x|\rfloor]$. The lengths of borders can be easily transformed to periods since x has period p if and only if it has a border of length $|x|-p$. ∎

THEOREM 1.5. *For every text $T \in [0\mathinner{.\,.}\sigma)^n$, there exists a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that answers* PERIODIC EXTENSION QUERIES *in $\mathcal{O}(1)$ time. The data structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Proof. Periodic Extension Queries can be answered using Prefix-Suffix Queries and LCE Queries in T and T^R. Given a fragment $x = T[\ell\mathinner{.\,.} r)$, we use a Prefix-Suffix Query to find the longest proper border of x provided that its length is at least $\frac{1}{2}|x|$. If no such border exists, we report that $\mathsf{run}(x) = \perp$. Otherwise, the length of the longest border yields the period $p = \mathrm{per}(x) \le \frac{1}{2}|x|$. In this case,

$$\mathsf{run}(x) = T[\ell - \mathrm{lcs}(T[0\mathinner{.\,.}\ell), T[0\mathinner{.\,.}\ell+p))\mathinner{.\,.}\ell + p + \mathrm{lcp}(T[\ell\mathinner{.\,.}n), T[\ell+p\mathinner{.\,.}n))),$$

where $\mathrm{lcs}(uv, w)$ denotes the length of their longest common suffix of strings v and w. ∎

9.3. Cyclic equivalence queries. Recall that, for a nonempty string $w \in \Sigma^n$, we define a string $\mathsf{rot}(w) = w[n-1]w[0]\cdots w[n-2]$. First, we prove that the sought set $\mathsf{Rot}(x,y) = \{j \in \mathbb{Z} : y = \mathsf{rot}^j(x)\}$ indeed forms an arithmetic progression.

Fact 9.4. *If* $\mathsf{Rot}(x,y) \ne \emptyset$*, then* $\mathsf{Rot}(x,y)$ *is an infinite arithmetic progression whose difference divides* $|x|$.

Proof. Let us note that if $j, j' \in \mathsf{Rot}(x,x)$, then $j - j' \in \mathsf{Rot}(x,x)$ and thus also $\gcd(j,j') \in \mathsf{Rot}(x,x)$. Consequently, $\mathsf{Rot}(x,x)$ consists of multiples of some integer m. Due to $|x| \in \mathsf{Rot}(x,x)$, this integer m is a divisor of $|x|$.

Next, observe that if $j \in \mathsf{Rot}(x,y)$, then $\mathsf{Rot}(x,y) = \{j + j' : j' \in \mathsf{Rot}(x,x)\}$. Hence, if $\mathsf{Rot}(x,y) \ne \emptyset$, then $\mathsf{Rot}(x,y)$ is an infinite arithmetic progression whose difference divides $|x|$. ∎

While answering Cyclic Equivalence Queries, we can assume that $|x| = |y|$; we denote the common length of x and y by d. Our query algorithm is based on the following characterization of $\mathsf{Rot}(x,y)$.

Observation 9.5. Let x, y be strings of common length d. For every $j \in [0\mathinner{.\,.}d]$, we have $j \in \mathsf{Rot}(x,y)$ if and only if $y[0\mathinner{.\,.}j) \cong x[d-j\mathinner{.\,.}d)$ and $y[j\mathinner{.\,.}d) \cong x[0\mathinner{.\,.}d-j)$.

Below, we provide an algorithm that computes $\mathsf{Rot}(x,y) \cap [\lceil\frac{d}{2}\rceil\mathinner{.\,.}d)$. By Observation 9.5, $j \in \mathsf{Rot}(x,y)$ if and only if $d - j \in \mathsf{Rot}(y,x)$, so running this algorithm for both (x,y) and (y,x) lets us retrieve $\mathsf{Rot}(x,y) \cap [1\mathinner{.\,.}d)$. This is sufficient to determine $\mathsf{Rot}(x,y)$ because an LCE Query lets us easily check if $x \cong y$, i.e., whether $0 \in \mathsf{Rot}(x,y)$, and Fact 9.4 yields $\mathsf{Rot}(x,y) = \{j \in \mathbb{Z} : j \bmod d \in \mathsf{Rot}(x,y) \cap [0\mathinner{.\,.}d)\}$.

By Observation 9.5, if $j \in [\lceil\frac{d}{2}\rceil\mathinner{.\,.}d) \cap \mathsf{Rot}(x,y)$, then the length-$j$ suffix of x matches a prefix of y. Since $d - 1 \le 2\lceil\frac{d}{2}\rceil - 1$, all lengths $j_0 < \cdots < j_{k-1}$ satisfying the latter condition form an arithmetic progression and can be retrieved with a single Prefix-Suffix Query. Moreover, Observation 9.5 yields $[\lceil\frac{d}{2}\rceil\mathinner{.\,.}d) \cap \mathsf{Rot}(x,y) = \{j_i : i \in [0\mathinner{.\,.}k) \text{ and } y[j_i\mathinner{.\,.}d) \cong x[0\mathinner{.\,.}d-j_i)\}$. Hence, it suffices to check for which indices i the suffix $y_i := y[j_i\mathinner{.\,.}d)$ of y matches a prefix of x. For this, we note that $(j_i)_{i=0}^{k-1}$ is a periodic progression in y: For each $i \in [1\mathinner{.\,.}k)$, the string $y[j_{i-1}\mathinner{.\,.}j_i)$ matches a suffix of x whose length is the difference of the arithmetic progression $(j_i)_{i=0}^{k-1}$. Hence, Lemma 1.14(a) lets us retrieve an integer interval consisting of indices i such that $j_i \in \mathsf{Rot}(x,y)$, and this interval can be easily transformed into an arithmetic progression of the corresponding values j_i. Consequently, the data structure of Theorem 1.2 (which already contains the component of Proposition 1.8 for LCE Queries and which can answer Prefix-Suffix Queries in $\mathcal{O}(1)$ time; cf. Theorem 1.4) can also answer Cyclic Equivalence Queries in $\mathcal{O}(1)$ time.

Theorem 1.6. *For every text* $T \in [0\mathinner{.\,.}\sigma)^n$*, there exists a data structure of size* $\mathcal{O}(n/\log_\sigma n)$ *that answers* Cyclic Equivalence Queries *in* $\mathcal{O}(1)$ *time. The data* *structure can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of T.*

Finally, let us note that the query algorithms behind Theorems 1.3–1.6 indeed access the text only through `PILLAR` operations. (Actually, even the `Access` method is not used; the characters of the strings are only accessed indirectly through LCE Queries and IPM Queries.) This implies Corollary 1.10.

9.4. Queries related to Lempel–Ziv compression. Substring Compression Queries are internal queries asking for a compressed representation of a substring or the (exact or approximate) size of this representation. This family of problems was introduced by Cormode and Muthukrishnan [34], and some of their results were later improved by Keller et al. [64]. Substring Compression Queries have a fairly direct motivation: Consider a server holding a long repetitive text T and clients asking for substrings of T (e.g., chunks that should be displayed). A limited capacity of the communication channel justifies compressing these substrings.

The aforementioned papers [34, 64] apply the classic LZ77 compression scheme [114]. Among other problems, they consider internal queries for the LZ factorization of a given fragment x and for the generalized LZ factorization of one fragment x in the context of another fragment y. The latter is defined as the part representing x in the LZ factorization of a string $y\#x$, where $\#$ is a special sentinel symbol not present in the text. A server can send such a generalized LZ factorization to a client who requests y and has previously received x.

Thus, in this section, we consider variants of LZ Substring Compression Queries. Let us first recall Lempel–Ziv LZ77 algorithm [114] and range successor queries that will be used in our solution.

LZ77 compression. Consider a string $w \in \Sigma^*$. We say that a fragment $w[\ell \mathinner{..} r)$ has a *previous occurrence* (or is a *previous fragment*) if $w[\ell \mathinner{..} r) \cong w[\ell' \mathinner{..} r')$ for some positions $\ell' < \ell$ and $r' < r$. The fragment $w[\ell \mathinner{..} r)$ has a *nonoverlapping previous occurrence* (or is a *nonoverlapping previous fragment*) if additionally $r' \le \ell$.

The Lempel–Ziv factorization $\mathrm{LZ}(w)$ is a factorization $w = f_1 \cdots f_k$ into fragments (called *phrases*) such that each phrase f_i is the longest previous fragment starting at position $|f_1 \cdots f_{i-1}|$ or a single letter if there is no such previous fragment. The nonoverlapping Lempel–Ziv factorization $\mathrm{LZ}_N(w)$ is defined analogously, allowing for nonoverlapping previous fragments only. Both factorizations (and several closely related variants) are useful for compression because a previous fragment can be represented using a reference to the previous occurrence (e.g., the positions of its endpoints).

Strings $w \in \Sigma^*$ are sometimes compressed with respect to a *context string* (or *dictionary string*) $v \in \Sigma^*$. Essentially, there are two ways to define the factorization $\mathrm{LZ}(w \mid v)$ of w with respect to v. In the *relative LZ factorization* [115, 81] $\mathrm{LZ}_R(w \mid v)$, each phrase is the longest fragment of w that starts at the given position and occurs in v (or a single letter if there is no such fragment). An alternative approach is to allow both substrings of v and previous fragments of w as phrases. This results in the *generalized LZ factorization*, denoted $\mathrm{LZ}_G(w \mid v)$; see [34, 64]. Equivalently, $\mathrm{LZ}_G(w \mid v)$ can be defined as the suffix of $\mathrm{LZ}(v\#w)$ corresponding to w, where $\#$ is a special symbol that is present neither in v nor in w. The previous fragments in the nonoverlapping generalized LZ factorization $\mathrm{LZ}_{NG}(w \mid v)$ must be nonoverlapping.

Example 9.6. Let $w = \texttt{aaaabaabaaaa}$ and $v = \texttt{baabab}$. We have

$$\mathrm{LZ}(w) = \texttt{a} \cdot \texttt{aaa} \cdot \texttt{b} \cdot \texttt{aabaa} \cdot \texttt{aa}, \qquad \mathrm{LZ}_N(w) = \texttt{a} \cdot \texttt{a} \cdot \texttt{aa} \cdot \texttt{b} \cdot \texttt{aab} \cdot \texttt{aaaa},$$

$$\mathrm{LZ}_R(w \mid v) = \mathtt{aa}\cdot\mathtt{aaba}\cdot\mathtt{aba}\cdot\mathtt{aa}\cdot\mathtt{a}, \qquad \mathrm{LZ}_G(w \mid v) = \mathtt{aa}\cdot\mathtt{aaba}\cdot\mathtt{abaa}\cdot\mathtt{aa},$$
$$\mathrm{LZ}_{GN}(w \mid v) = \mathtt{aa}\cdot\mathtt{aaba}\cdot\mathtt{aba}\cdot\mathtt{aaa}.$$

Range successor queries. We define the *successor* of an integer t in a set A as $\mathrm{succ}_A(t) = \min\{z \in A : z > t\}$. Successor queries on a range $A[\ell\,..\,r]$ of an array A are defined as follows.

RANGE SUCCESSOR QUERIES (RANGE NEXT VALUE QUERIES)
Input: An array A of n integers.
Queries Given a range $[\ell\,..\,r]$ and an integer t, compute $\mathrm{succ}_{A[\ell..r]}(t)$ (and an index $j \in [\ell\,..\,r]$ such that $A[j] = \mathrm{succ}_{A[\ell..r]}(t)$, if any).

The following three trade-offs describe the current state of the art for such queries.

PROPOSITION 9.7. *For any constant $\varepsilon > 0$ and the functions S_{rsucc}, Q_{rsucc}, and C_{rsucc} specified below, there is a data structure of size $S_{rsucc}(n)$ that answers range successor queries in $Q_{rsucc}(n)$ time and can be constructed in $C_{rsucc}(n)$ time:*
(a) $S_{rsucc}(n) = \mathcal{O}(n)$, $Q_{rsucc}(n) = \mathcal{O}(\log^{\varepsilon} n)$, *and* $C_{rsucc}(n) = \mathcal{O}(n\sqrt{\log n})$ [93, 17];
(b) $S_{rsucc}(n) = \mathcal{O}(n \log\log n)$, $Q_{rsucc}(n) = \mathcal{O}(\log\log n)$, *and* $C_{rsucc}(n) = \mathcal{O}(n\sqrt{\log n})$ [113, 48];
(c) $S_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$, $Q_{rsucc}(n) = \mathcal{O}(1)$, *and* $C_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ [37].

LZ substring compression queries. We consider the following types of queries.

(NONOVERLAPPING) LZ SUBSTRING COMPRESSION QUERIES
Given a fragment x of T, compute the (nonoverlapping) LZ factorization of x, i.e., $LZ(x)$ (resp., $LZ_N(x)$).

RELATIVE LZ SUBSTRING COMPRESSION QUERIES
Given two fragments x and y of T, compute the relative LZ factorization of x with respect to y, i.e., $LZ_R(x \mid y)$.

GENERALIZED (NONOVERLAPPING) LZ SUBSTRING COMPRESSION QUERIES
Given two fragments x and y of T, compute the generalized (nonoverlapping) LZ factorization of x with respect to y, i.e., $LZ_G(x \mid y)$ (resp., $LZ_{GN}(x \mid y)$).

Our query algorithms heavily rely on the results of Keller et al. [64] for LZ SUBSTRING COMPRESSION QUERIES and GENERALIZED LZ SUBSTRING COMPRESSION QUERIES. The main improvement is a more efficient solution for the following auxiliary problem.

Bounded Longest Common Prefix (LCP) Queries
Given two fragments x and y of T, find the longest prefix p of x which occurs in y.

The other, easier auxiliary problem defined in [64] is used as a black box.

Interval Longest Common Prefix (LCP) Queries
Given a fragment x of T and an interval $[\ell\,..\,r]$ of positions in T, find the longest prefix p of x which occurs in T at some position within $[\ell\,..\,r]$.

The data structure for Interval Longest Common Prefix (LCP) Queries uses range successor queries, so we state the complexity in an abstract form. This convention gets propagated to further results in this section.

Lemma 9.8 ([64]). *For a text T of length n, there exists a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that answers* Interval Longest Common Prefix (LCP) Queries *in $\mathcal{O}(Q_{rsucc}(n))$ time. The data structure can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.*

As observed in [64], the decision version of IPM Queries easily reduces to Interval Longest Common Prefix (LCP) Queries. For $x = T[\ell_x\,..\,r_x)$ and $y = T[\ell_y\,..\,r_y)$, it suffices to check if the longest prefix of x occurring at some position in $[\ell_y\,..\,r_y - |x|]$ of T is x itself.

Corollary 9.9 ([64]). *For a text T of length n, there exists a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that, given fragments x, y of T, can decide in $\mathcal{O}(Q_{rsucc}(n))$ time whether x occurs in y. The data structure can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.*

We proceed with our solution for Bounded LCP Queries. Let $x = T[\ell_x\,..\,r_x)$ and $y = T[\ell_y\,..\,r_y)$. First, we search for the largest k such that the prefix of x of length 2^k (i.e., $T[\ell_x\,..\,\ell_x + 2^k)$) occurs in y. We use a variant of the binary search involving exponential search (also called galloping search), which requires $\mathcal{O}(\log K)$ steps, where K is the optimal value of k. At each step, for a fixed k, we need to decide if $T[\ell_x\,..\,\ell_x + 2^k)$ occurs in y. This can be done in $\mathcal{O}(Q_{rsucc}(n))$ time using Corollary 9.9. At this point, we have an integer K such that the optimal prefix p has length $|p| \in [2^K\,..\,2^{K+1})$. The running time is $\mathcal{O}(Q_{rsucc}(n) \log K) = \mathcal{O}(Q_{rsucc}(n) \log\log |p|)$ so far.

Let p' be the prefix obtained from an Interval Longest Common Prefix Query for x and $[\ell_y\,..\,r_y - 2^{K+1}]$. We have $|p'| < 2^{K+1}$, and thus the occurrence of p' starting in $[\ell_y\,..\,r_y - 2^{K+1}]$ lies within y. Consequently, $|p| \geq |p'|$; moreover, if p occurs at a position within $[\ell_y\,..\,r_y - 2^{K+1}]$, then $p = p'$.

The other possibility is that p only occurs near the end of y, i.e., within the suffix of y of length $2^{K+1} - 1$, which we denote as y'. We use a similar approach as for Prefix-Suffix Queries with $d = 2^K$ to detect p in this case. We define x' as the prefix of x of length 2^K. An occurrence of p must start with an occurrence of x', so we find all occurrences of x' in y'. If there are no such occurrences, we conclude that $p = p'$.

Otherwise, we define y_i as the suffix of y starting with the ith occurrence of x; see Figure 15. Next, we apply Lemma 1.14(b) to compute $\max_i \operatorname{lcp}(y_i, x)$. By the discussion above, this must be the length of the longest prefix of x that occurs in y'. We compare its length to $|p'|$ and choose the final answer p as the longer of the two candidates.

Thus, the data structure for IPM QUERIES, accompanied by the components of Lemma 9.8, Corollary 9.9, and Proposition 1.8, yields the following result.

THEOREM 1.7. *For every text T of length n over an alphabet $[0\,..\,n^{\mathcal{O}(1)})$, there exists a data structure of size $\mathcal{O}(n+S_{rsucc}(n))$ that answers* BOUNDED LCP QUERIES *in $\mathcal{O}(Q_{rsucc}(n)\log\log|p|)$ time. The data structure can be constructed in $\mathcal{O}(n+C_{rsucc}(n))$ time.*

Finally, we generalize the approach of [64] to support multiple types of LZ SUBSTRING COMPRESSION QUERIES using Theorem 1.7 to improve the running time.

THEOREM 9.10. *For every text T of length n, there is a data structure of size $\mathcal{O}(n+S_{rsucc}(n))$ that answers*
(A) NONOVERLAPPING LZ SUBSTRING COMPRESSION QUERIES*,*
(B) RELATIVE LZ SUBSTRING COMPRESSION QUERIES*,*
(C) GENERALIZED LZ SUBSTRING COMPRESSION QUERIES*, and*
(D) GENERALIZED NONOVERLAPPING LZ SUBSTRING COMPRESSION QUERIES*,*
each in $\mathcal{O}\big(F\cdot Q_{rsucc}(n)\log\log\frac{|x|}{F}\big)$ time, where F is the number of phrases reported. The data structure can be constructed in $\mathcal{O}(n+C_{rsucc}(n))$ time.

Proof. Let $x=T[\ell_x\,..\,r_x)$, and suppose that we have already factorized $x'=T[\ell_x\,..\,m)$; i.e., the next phrase needs to be a prefix of $x''=T[m\,..\,r_x)$. Depending on the factorization type, it is chosen among the longest prefix of x'' that is a previous fragment of x (i.e., has an occurrence starting within $[\ell_x\,..\,m)$), the longest prefix of x'' that is a nonoverlapping previous fragment of x (i.e., occurs in x'), or the longest prefix of x'' that occurs in y. The first case reduces to an INTERVAL LONGEST COMMON PREFIX QUERY, while the latter two reduce to BOUNDED LCP QUERIES. For each factorization type, we compute the relevant candidates and choose the longest one as the phrase; if there are no valid candidates, the next phrase is a single letter, i.e., $T[m\,..\,m]$.

Thus, regardless of the factorization type, we report each phrase f_i of the factorization $x=f_1\cdots f_F$ in $\mathcal{O}(Q_{rsucc}(n)\log\log|f_i|)$ time. This way, the total running time is $\mathcal{O}\big(\sum_{i=1}^{F}Q_{rsucc}(n)\log\log|f_i|\big)$, which is $\mathcal{O}\big(F\cdot Q_{rsucc}(n)\log\log\frac{|x|}{F}\big)$ due to Jensen's inequality applied to the concave $\log\log$ function. ∎

Let us note that in the case of ordinary LZ SUBSTRING COMPRESSION QUERIES, the approach presented in Theorem 9.10 would result in $\mathcal{O}(F\cdot Q_{rsucc}(n))$ query time because only INTERVAL LONGEST COMMON PREFIX (LCP) QUERIES would be used; this is exactly the algorithm for LZ SUBSTRING COMPRESSION QUERIES provided in [64].

Hence, despite our improvements, there is still an overhead for using variants of the LZ factorization other than the standard one. Nevertheless, the overhead disappears if we use the state-of-the-art $\mathcal{O}(n)$-size data structure for range successor queries. This is because the $\mathcal{O}(\log^{\varepsilon} n)$ time complexity lets us hide $\log^{o(1)} n$ factors by choosing a slightly greater ε. Formally, Theorem 9.10 and Proposition 9.7 yield the following result.

COROLLARY 9.11. *For every text T of length n over an alphabet $[0\,..\,n^{\mathcal{O}(1)})$ and constant $\varepsilon>0$, there is a data structure of size $\mathcal{O}(n)$ that answers* BOUNDED LCP

QUERIES *in* $\mathcal{O}(\log^{\varepsilon} n)$ *time and* LZ SUBSTRING COMPRESSION QUERIES *(for all five factorization types defined above) in* $\mathcal{O}(\log^{\varepsilon} n)$ *time per phrase reported. Moreover, the data structure can be constructed in* $\mathcal{O}(n\sqrt{\log n})$ *time.*

Appendix A. Auxiliary proofs.

Observation 3.3. For every $k \in \mathbb{Z}_{\geq 0}$, we have $\alpha_{k+1} \leq 16\lambda_k$.

Proof. We denote $Z = \left\{\left\lfloor\frac{k}{2}\right\rfloor, \left\lfloor\frac{k-1}{2}\right\rfloor\right\}$ and observe that $7 \cdot \mu^{z+1} = 8\,\mu^z$ for any $z \in \mathbb{Z}_{\geq 0}$ and $\frac{1}{\mu-1} = 7$. We have

$$\begin{aligned}\alpha_{k+1} &= 1 + \sum_{t=0}^{k} \lfloor \lambda_t \rfloor \leq 1 + \sum_{t=0}^{k} \mu^{\lfloor t/2 \rfloor} = 1 + \sum_{z \in Z}\sum_{t=0}^{z} \mu^t = 1 + \sum_{z\in Z} 7 \cdot (\mu^{z+1} - 1)\\ &< 8\lambda_k + 8\lambda_{k-1} \leq 16\lambda_k.\end{aligned}$$
∎

LEMMA A.1. APPROXIMATE MAXIMUM DIRECTED CUT *problem can be solved in linear time.*

Proof. First, we preprocess G so that each $v \in V$ stores both incoming and outgoing arcs. For $A, B \subseteq V$, we denote $E(A,B) \subseteq E$ to be the set of arcs leading from A to B; recall that the goal is to make sure that $|E(L,R)| \geq \frac{1}{4}|E|$. Given $v \in V$ and $A \subseteq V$, define $\deg^+_A(v) := |E(\{v\}, A)|$ and $\deg^-_A(v) := |E(A, \{v\})|$.

We maintain a partition $V = L \cup M \cup R$ into three disjoint classes. Initially, $M = V$, and, as long as $M \neq \emptyset$, we pick an arbitrary vertex $v \in M$ and move v to L or R, depending on whether

$$2\deg^+_R(v) + \deg^+_M(v) \geq 2\deg^-_L(v) + \deg^-_M(v).$$

This decision can be implemented in $\mathcal{O}(1 + \deg^+_V(v) + \deg^-_V(v))$ time, which yields a total running time of $\mathcal{O}(|V| + |E|)$.

As for correctness, we shall prove that

$$\Phi := 4|E(L,R)| + 2|E(L,M)| + 2|E(M,R)| + |E(M,M)|$$

cannot decrease throughout the algorithm. Consider the effect of moving v from M to L on the four terms of Φ (recall that there are no self-loops $v \to v$):

- $|E(L,R)|$ increases by $\deg^+_R(v)$;
- $|E(L,M)|$ increases by $\deg^+_M(v)$ and decreases by $\deg^-_L(v)$;
- $|E(M,R)|$ decreases by $\deg^+_R(v)$;
- $|E(M,M)|$ decreases by $\deg^+_M(v)$ and decreases by $\deg^-_M(v)$.

Overall, Φ increases by

$$\begin{aligned}&4 \cdot \deg^+_R(v) + 2 \cdot (\deg^+_M(v) - \deg^-_L(v)) + 2 \cdot (-\deg^+_R(v)) + (-\deg^+_M(v) - \deg^-_M(v))\\ &\quad = 2\deg^+_R(v) + \deg^+_M(v) - 2\deg^-_L(v) - \deg^-_M(v),\end{aligned}$$

and this quantity is nonnegative when the algorithm decides to move v to L.

Similarly, if v is moved from M to R, then Φ does not decrease. On the end of the algorithm, we have $\Phi = 4|E(L,R)|$ due to $M = \emptyset$, whereas, initially, $\Phi = |E(M,M)| = |E|$ due to $M = V$. Since Φ is noncecreasing, we conclude that $4|E(L,R)| \geq |E|$ holds as claimed. ∎

Appendix B. Information-theoretic bounds on the set of occurrences. By $Occ(x,y)$, we denote the set of starting positions of all occurrences of string x in string y. The following result provides matching worst-case bounds for the encoding size of $Occ(x,y)$.

PROPOSITION B.1. *For all integers $1 \le m \le n$ and alphabets Σ of size at least two, in order to encode $Occ(x,y)$ for $x \in \Sigma^m$ and $y \in \Sigma^n$,*

$$\Theta\left(\frac{n}{m}\log\min(m+1, n-m+2)\right)$$

bits are sufficient and, in the worst case, necessary.

Proof. Let us first prove the upper bound. If $n < 2m$, then $Occ(x,y) \subseteq [0\,..\,n-m]$ forms a (possibly empty) arithmetic progression; such a progression can be encoded using at most three elements of $[0\,..\,n-m]$, i.e., using at most $3\log(n-m+2)$ bits. If $n \ge 2m$, then one can decompose y into $\lfloor n/m \rfloor$ fragments of length $2m-1$ contained in y, with overlaps of at least $m-1$ characters between the subsequent fragments. The occurrences of x within each of these fragments can be encoded in $3\log(m+1)$ bits, for a total of $3n/m \cdot \log(m+1)$ bits.

As for the lower bound, suppose that $\mathtt{0}$ and $\mathtt{1}$ are two distinct characters in Σ, and consider a pattern $x = \mathtt{0}^m$. If $n \le 2m$, consider texts of the form $y_i = \mathtt{1}^i\mathtt{0}^{n-i}$ for $i \in [0\,..\,n-m+1]$, and observe that the sets $Occ(x,y_i) = [i\,..\,n-m]$ are pairwise different. Hence, encoding $Occ(x,y)$ requires $\log(n-m+2)$ bits in the worst case. If $n \ge 2m$, let $b = \lfloor n/(2m) \rfloor$, and, for every sequence $s \in [0\,..\,m]^b$, define a sequence $y_s = (\mathtt{1}^{s[0]}\mathtt{0}^m\mathtt{1}^{m-s[0]}) \cdot (\mathtt{1}^{s[1]}\mathtt{0}^m\mathtt{1}^{m-s[1]}) \cdots (\mathtt{1}^{s[b-1]}\mathtt{0}^m\mathtt{1}^{m-s[b-1]}) \cdot \mathtt{1}^{n-2mb}$. Observe that the sets $Occ(x,y_s)$ are pairwise distinct since $Occ(x,y_s) \cap [2mi\,..\,2m(i+1)] = \{s[i]\}$ holds for each $i \in [0\,..\,b)$ and $s \in [0\,..\,m]^b$. Thus, encoding $Occ(x,y)$ requires $b\log(m+1) \ge \frac{n}{4m}\log(m+1)$ bits in the worst case. ∎

Appendix C. Implementations of Corollary 1.10. In this section, we obtain efficient implementations of the queries of Corollary 1.10 in the dynamic, fully compressed, and quantum settings.

C.1. Dynamic setting. Let $\mathcal{X}$ be a growing collection of nonempty persistent strings; it is initially empty and then undergoes updates by means of the following operations:

- $\mathtt{Makestring}(u)$: Insert a nonempty string u to $\mathcal{X}$.
- $\mathtt{Concat}(u,v)$: Insert string uv to $\mathcal{X}$ for $u,v \in \mathcal{X}$.
- $\mathtt{Split}(u,i)$: Insert $u[0\,..\,i)$ and $u[i\,..\,|u|)$ to $\mathcal{X}$ for $u \in \mathcal{X}$ and $i \in [0\,..\,|u|)$.

By N, we denote an upper bound on the total length of all strings in $\mathcal{X}$ throughout all updates executed by an algorithm. As shown in [30] using [51], a collection $\mathcal{X}$ of nonempty persistent strings of total length N can be dynamically maintained with operations $\mathtt{Makestring}(u)$, $\mathtt{Concat}(u,v)$, and $\mathtt{Split}(u,i)$, requiring time $\mathcal{O}(\log N + |u|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, so that $\mathtt{PILLAR}$ operations can be performed in time $\mathcal{O}(\log^2 N)$. All stated time complexities hold with probability $1 - 1/N^{\Omega(1)}$. Moreover, Kempa and Kociumaka [67, sect. 8 in the arXiv version] presented an alternative deterministic implementation which supports operations $\mathtt{Makestring}(u)$, $\mathtt{Concat}(u,v)$, and $\mathtt{Split}(u,i)$ in time $\mathcal{O}(|u|\log^{\mathcal{O}(1)}\log N)$, $\mathcal{O}(\log|uv|\log^{\mathcal{O}(1)}\log N)$, and $\mathcal{O}(\log|u|\log^{\mathcal{O}(1)}\log N)$, respectively, so that $\mathtt{PILLAR}$ operations can be performed in time $\mathcal{O}(\log N \log^{\mathcal{O}(1)}\log N)$. With these implementations, we obtain the following result.

THEOREM C.1 (dynamic setting). *A collection $\mathcal{X}$ of nonempty persistent strings of total length N can be dynamically maintained with operations* `Makestring`(u), `Concat`(u,v), *and* `Split`(u,i), *requiring time $\mathcal{O}(\log N + |u|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, so that we can answer* PREFIX-SUFFIX QUERIES, 2-PERIOD QUERIES, *and* CYCLIC EQUIVALENCE QUERIES *on strings from $\mathcal{X}$ in $\mathcal{O}(\log^2 N)$ time and* PERIOD QUERIES *in $\mathcal{O}(\log^3 N)$ time. All stated time complexities hold with probability $1-1/N^{\Omega(1)}$. Randomization can be avoided at the cost of a $\log^{\mathcal{O}(1)} \log N$ multiplicative factor in all the update times, with* PREFIX-SUFFIX QUERIES, 2-PERIOD QUERIES, *and* CYCLIC EQUIVALENCE QUERIES *on strings from $\mathcal{X}$ answered in $\mathcal{O}(\log N \log^{\mathcal{O}(1)} \log N)$ time and* PERIOD QUERIES *in $\mathcal{O}(\log^2 N \log^{\mathcal{O}(1)} \log N)$ time.*

C.2. Fully compressed setting. A *straight-line grammar* is a context-free grammar G that consists of a set Σ of terminals and a set $N_G = \{A_1, \ldots, A_n\}$ of nonterminals such that each $A_i \in N_G$ is associated with a unique production rule $A_i \to f_G(A_i) \in (\Sigma \cup \{A_j : j < i\})^*$. We can assume without loss of generality that G is a *straight-line program*; that is, each production rule is of the form $A \to BC$ for some symbols B and C. Every symbol $A \in S_G := N_G \cup \Sigma$ generates a unique string, which we denote by $\mathsf{val}(A) \in \Sigma^*$. The string $\mathsf{val}(A)$ can be obtained from A by repeatedly replacing each nonterminal with its production. We say that G generates $\mathsf{val}(G) := \mathsf{val}(A_n)$.

In the fully compressed setting, given a collection $\mathcal{X}$ of straight-line programs of total size n generating strings of total length N, each `PILLAR` operation can be performed in $\mathcal{O}(\log^2 N \log\log N)$ time after an $\mathcal{O}(n \log N)$-time preprocessing [30].

THEOREM C.2 (fully compressed setting). *Let G_X, G_Y denote straight-line programs of total size n generating strings X, Y, respectively, of total length N. For strings X and Y, we can answer a* PREFIX-SUFFIX QUERY, *a* 2-PERIOD QUERY, *and a* CYCLIC EQUIVALENCE QUERY *in $\mathcal{O}(n \log N + \log^2 N \log\log N)$ time and a* PERIOD QUERY *in $\mathcal{O}(n \log N + \log^3 N \log\log N)$ time.*

In particular, Theorem C.2 can be compared with the algorithm of I et al. [56] that, with the improvement of Ganardi, Jeż, and Lohrey [47], computes all the periods of a string of length N generated by a straight-line program of size n in $\mathcal{O}(n^2 \log N)$ time. (We note that $N \le 2^n$.)

C.3. Quantum setting. We say that an algorithm on an input of size n succeeds *with high probability* if the success probability can be made at least $1 - 1/n^c$ for any desired constant $c > 1$. In what follows, we assume that the input strings can be accessed in a quantum query model [5, 22]. We are interested in the time complexity of quantum algorithms.

In the quantum setting, LCE QUERIES can be answered in $\tilde{\mathcal{O}}(\sqrt{n})$[5] time with high probability [61, Observation 2.3]. IPM QUERIES can be answered using the work of Hariharan and Vinay [54].

LEMMA C.3. IPM QUERIES *can be answered in $\tilde{\mathcal{O}}(\sqrt{n})$ time with high probability in the quantum model.*

Proof. Hariharan and Vinay [54] (see also [110] for an algorithm with improved polylogarithmic factors) showed how to determine whether a given pattern of length m occurs in a given text of length n in $\tilde{\mathcal{O}}(\sqrt{n} + \sqrt{m})$ time in the quantum model with high probability. If the answer is positive, then the algorithm can return the leftmost

[5]The $\tilde{\mathcal{O}}$ notation suppresses any $\log^{\mathcal{O}(1)} n$ factors.

occurrence. By Remark 1.1, this is sufficient to answer an IPM QUERY in $\tilde{\mathcal{O}}(\sqrt{n})$ time. ◻

All other PILLAR operations are performed trivially in $\mathcal{O}(1)$ quantum time. Thus, while all PILLAR operations can be implemented in $\mathcal{O}(1)$ time after $\mathcal{O}(n/\log_\sigma n)$-time preprocessing in the standard setting by a classic algorithm, in the quantum setting, all PILLAR operations can be implemented in $\tilde{\mathcal{O}}(\sqrt{n})$ quantum time *with no preprocessing.* We obtain the following result.

THEOREM C.4 (quantum setting). *Let X and Y be strings of total length n. For strings X and Y, we can answer a* PREFIX-SUFFIX QUERY, *a* PERIOD QUERY, *and a* CYCLIC EQUIVALENCE QUERY *in $\tilde{\mathcal{O}}(\sqrt{n})$ time with high probability in the quantum model.*

Wang and Ying [110] proposed a bounded-error quantum algorithm answering 2-PERIOD QUERIES in $\tilde{\mathcal{O}}(\sqrt{n})$ time. Akmal and Jin [4] showed how to compute the lexicographically minimal rotation of a string in $n^{1/2+o(1)}$ time in the quantum model. With LCE QUERIES, this implies a $n^{1/2+o(1)}$-time algorithm for CYCLIC EQUIVALENCE QUERIES, slightly slower than the algorithm of Theorem C.4.

Acknowledgments. The authors wish to thank Dominik Kempa for helpful discussions regarding synchronizing sets and Moshe Lewenstein for a suggestion to work on the generalized substring compression problem.

REFERENCES

[1] P. ABEDIN, A. GANGULY, W. HON, K. MATSUDA, Y. NEKRICH, K. SADAKANE, R. SHAH, AND S. V. THANKACHAN, *A linear-space data structure for range-LCP queries in poly-logarithmic time*, Theoret. Comput. Sci., 822 (2020), pp. 15–22, https://doi.org/10.1016/j.tcs.2020.04.009.

[2] P. ABEDIN, A. GANGULY, S. P. PISSIS, AND S. V. THANKACHAN, *Efficient data structures for range shortest unique substring queries*, Algorithms, 13 (2020), 276, https://doi.org/10.3390/a13110276.

[3] A. V. AHO AND M. J. CORASICK, *Efficient string matching: An aid to bibliographic search*, Commun. ACM, 18 (1975), pp. 333–340, https://doi.org/10.1145/360825.360855.

[4] S. AKMAL AND C. JIN, *Near-optimal quantum algorithms for string problems*, Algorithmica, 85 (2023), pp. 2260–2317, https://doi.org/10.1007/S00453-022-01092-X.

[5] A. AMBAINIS, *Quantum query algorithms and lower bounds*, in Classical and New Paradigms of Computation and Their Complexity Hierarchies, B. Löwe, B. Piwinger, and T. Räsch, eds., Springer-Verlag, Berlin, 2004, pp. 15–32, https://doi.org/10.1007/978-1-4020-2776-5_2.

[6] A. AMIR, M. AMIT, G. M. LANDAU, AND D. SOKOL, *Period recovery of strings over the Hamming and edit distances*, Theoret. Comput. Sci., 710 (2018), pp. 2–18, https://doi.org/10.1016/j.tcs.2017.10.026.

[7] A. AMIR, A. APOSTOLICO, G. M. LANDAU, A. LEVY, M. LEWENSTEIN, AND E. PORAT, *Range LCP*, J. Comput. System Sci., 80 (2014), pp. 1245–1253, https://doi.org/10.1016/j.jcss.2014.02.010.

[8] A. AMIR, I. BONEH, P. CHARALAMPOPOULOS, AND E. KONDRATOVSKY, *Repetition detection in a dynamic string*, in 27th Annual European Symposium on Algorithms, ESA 2019, LIPIcs 144, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2019, pp. 5:1–5:18, https://doi.org/10.4230/LIPIcs.ESA.2019.5.

[9] A. AMIR, A. BUTMAN, E. KONDRATOVSKY, A. LEVY, AND D. SOKOL, *Multidimensional period recovery*, Algorithmica, 84 (2022), pp. 1490–1510, https://doi.org/10.1007/s00453-022-00926-y.

[10] A. AMIR, P. CHARALAMPOPOULOS, S. P. PISSIS, AND J. RADOSZEWSKI, *Dynamic and internal longest common substring*, Algorithmica, 82 (2020), pp. 3707–3743, https://doi.org/10.1007/s00453-020-00744-0.

[11] A. AMIR, G. M. LANDAU, M. LEWENSTEIN, AND D. SOKOL, *Dynamic text and static pattern matching*, ACM Trans. Algorithms, 3 (2007), 19, https://doi.org/10.1145/1240233.1240242.

[12] A. Amir, G. M. Landau, S. Marcus, and D. Sokol, *Two-dimensional maximal repetitions*, Theoret. Comput. Sci., 812 (2020), pp. 49–61, https://doi.org/10.1016/j.tcs.2019.07.006.

[13] A. Amir, M. Lewenstein, and S. V. Thankachan, *Range LCP queries revisited*, in 22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015, LNCS 9309, Springer-Verlag, Berlin, 2015, pp. 350–361, https://doi.org/10.1007/978-3-319-23826-5_33.

[14] M. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya, *Wavelet trees meet suffix trees*, in 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, Philadelphia, 2015, pp. 572–591, https://doi.org/10.1137/1.9781611973730.39.

[15] G. Badkobeh, P. Charalampopoulos, D. Kosolobov, and S. P. Pissis, *Internal shortest absent word queries in constant time and linear space*, Theoret. Comput. Sci., 922 (2022), pp. 271–282, https://doi.org/10.1016/j.tcs.2022.04.029.

[16] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta, *The "runs" theorem*, SIAM J. Comput., 46 (2017), pp. 1501–1514, https://doi.org/10.1137/15M1011032.

[17] D. Belazzougui and S. J. Puglisi, *Range predecessor and Lempel-Ziv parsing*, in 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, SIAM, Philadelphia, 2016, pp. 2053–2071, https://doi.org/10.1137/1.9781611974331.ch143.

[18] O. Ben-Kiki, P. Bille, D. Breslauer, L. Gąsieniec, R. Grossi, and O. Weimann, *Towards optimal packed string matching*, Theoret. Comput. Sci., 525 (2014), pp. 111–129, https://doi.org/10.1016/j.tcs.2013.06.013.

[19] O. Birenzwige, S. Golan, and E. Porat, *Locally consistent parsing for text indexing in small space*, in 31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, SIAM, Philadelphia, 2020, pp. 607–626, https://doi.org/10.1137/1.9781611975994.37.

[20] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Commun. ACM, 20 (1977), pp. 762–772, https://doi.org/10.1145/359842.359859.

[21] D. Breslauer and Z. Galil, *Finding all periods and initial palindromes of a string in parallel*, Algorithmica, 14 (1995), pp. 355–366, https://doi.org/10.1007/BF01294132.

[22] H. Buhrman and R. de Wolf, *Complexity measures and decision tree complexity: A survey*, Theoret. Comput. Sci., 288 (2002), pp. 21–43, https://doi.org/10.1016/S0304-3975(01)00144-X.

[23] S. Butakov and V. Scherbinin, *The toolbox for local and global plagiarism detection*, Comput. Educ., 52 (2009), pp. 781–788, https://doi.org/10.1016/j.compedu.2008.12.001.

[24] T. M. Chan, S. Golan, T. Kociumaka, T. Kopelowitz, and E. Porat, *Approximating text-to-pattern Hamming distances*, in 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, ACM, New York, 2020, pp. 643–656, https://doi.org/10.1145/3357713.3384266.

[25] P. Charalampopoulos, T. Kociumaka, M. Mohamed, J. Radoszewski, W. Rytter, and T. Waleń, *Internal dictionary matching*, Algorithmica, 83 (2021), pp. 2142–2169, https://doi.org/10.1007/s00453-021-00821-y.

[26] P. Charalampopoulos, T. Kociumaka, S. P. Pissis, and J. Radoszewski, *Faster algorithms for longest common substring*, in 29th Annual European Symposium on Algorithms, ESA 2021, LIPIcs 204, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2021, pp. 30:1–30:17, https://doi.org/10.4230/LIPIcs.ESA.2021.30.

[27] P. Charalampopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, and W. Zuba, *Circular pattern matching with k mismatches*, J. Comput. System Sci., 115 (2021), pp. 73–85, https://doi.org/10.1016/j.jcss.2020.07.003.

[28] P. Charalampopoulos, T. Kociumaka, J. Radoszewski, S. P. Pissis, W. Rytter, T. Waleń, and W. Zuba, *Approximate circular pattern matching*, in 30th Annual European Symposium on Algorithms, ESA 2022, LIPIcs 244, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2022, pp. 35:1–35:19, https://doi.org/10.4230/LIPIcs.ESA.2022.35.

[29] P. Charalampopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, and W. Zuba, *Efficient enumeration of distinct factors using package representations*, in 27th International Symposium on String Processing and Information Retrieval, SPIRE 2020, LNCS 12303, Springer-Verlag, Berlin, 2020, pp. 247–261, https://doi.org/10.1007/978-3-030-59212-7_18.

[30] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz, *Faster approximate pattern matching: A unified approach*, in 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, IEEE, New York, 2020, pp. 978–989, https://doi.org/10.1109/FOCS46700.2020.00095.

[31] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz, *Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed*

monoid of permutation matrices, in 63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, IEEE, New York, 2022, pp. 698–707, https://doi.org/10.1109/FOCS54457.2022.00072.

[32] P. Charalampopoulos, S. P. Pissis, J. Radoszewski, W. Rytter, T. Waleń, and W. Zuba, *Approximate circular pattern matching under edit distance*, in 41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, LIPIcs 289, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2024, pp. 24:1–24:22, https://doi.org/10.4230/LIPICS.STACS.2024.24.

[33] K. T. Chen, R. H. Fox, and R. C. Lyndon, *Free differential calculus,* IV. *The quotient groups of the lower central series*, Ann. Math., 68 (1958), pp. 81–95, https://doi.org/10.2307/1970044.

[34] G. Cormode and S. Muthukrishnan, *Substring compression problems*, in 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, SIAM, Philadelphia, 2005, pp. 321–330, http://dl.acm.org/citation.cfm?id=1070432.1070478.

[35] M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on Strings*, Cambridge University Press, Cambridge, 2007, https://doi.org/10.1017/cbo9780511546853.

[36] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń, *Extracting powers and periods in a word from its runs structure*, Theoret. Comput. Sci., 521 (2014), pp. 29–41, https://doi.org/10.1016/j.tcs.2013.11.018.

[37] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, G. Tischler, and T. Waleń, *Improved algorithms for the range next value problem and applications*, Theoret. Comput. Sci., 434 (2012), pp. 23–34, https://doi.org/10.1016/j.tcs.2012.02.015.

[38] M. Crochemore, C. S. Iliopoulos, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, and W. Zuba, *Internal quasiperiod queries*, in 27th International Symposium on String Processing and Information Retrieval, SPIRE 2020, LNCS 12303, Springer-Verlag, Berlin, 2020, pp. 60–75, https://doi.org/10.1007/978-3-030-59212-7_5.

[39] M. Crochemore and W. Rytter, *Jewels of Stringology*, World Scientific, River Edge, NJ, 2003, https://doi.org/10.1142/4838.

[40] D. Das, T. Kociumaka, and B. Saha, *Improved approximation algorithms for Dyck edit distance and RNA folding*, in 49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, LIPIcs 229, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2022, pp. 49:1–49:20, https://doi.org/10.4230/LIPIcs.ICALP.2022.49.

[41] R. Das, M. He, E. Kondratovsky, J. I. Munro, and K. Wu, *Internal masked prefix sums and its connection to fully internal measurement queries*, in 29th International Symposium on String Processing and Information Retrieval, SPIRE 2022, LNCS 13617, Springer-Verlag, Berlin, 2022, pp. 217–232, https://doi.org/10.1007/978-3-031-20643-6_16.

[42] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, *KMC* 2*: Fast and resource-frugal k-mer counting*, Bioinformatics, 31 (2015), pp. 1569–1576, https://doi.org/10.1093/bioinformatics/btv022.

[43] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, *On the sorting-complexity of suffix tree construction*, J. ACM, 47 (2000), pp. 987–1011, https://doi.org/10.1145/355541.355547.

[44] M. Farach-Colton and S. Muthukrishnan, *Perfect hashing for strings: Formalization and algorithms*, in 7th Annual Symposium on Combinatorial Pattern Matching, CPM 1996, LNCS 1075, Springer-Verlag, Berlin, 1996, pp. 130–140, https://doi.org/10.1007/3-540-61258-0_11.

[45] H. Fernau, F. Manea, R. Mercas, and M. L. Schmid, *Pattern matching with variables: Efficient algorithms and complexity results*, ACM Trans. Comput. Theory, 12 (2020), pp. 6:1–6:37, https://doi.org/10.1145/3369935.

[46] N. J. Fine and H. S. Wilf, *Uniqueness theorems for periodic functions*, Proc. Amer. Math. Soc., 16 (1965), pp. 109–114, https://doi.org/10.2307/2034009.

[47] M. Ganardi, A. Jeż, and M. Lohrey, *Balancing straight-line programs*, J. ACM, 68 (2021), pp. 27:1–27:40, https://doi.org/10.1145/3457389.

[48] Y. Gao, M. He, and Y. Nekrich, *Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing*, in 28th Annual European Symposium on Algorithms, ESA 2020, LIPIcs 173, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2020, pp. 54:1–54:18, https://doi.org/10.4230/LIPIcs.ESA.2020.54.

[49] P. Gawrychowski, *Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic*, in 19th Annual European Symposium on Algorithms, ESA 2011, LNCS 6942, Springer-Verlag, Berlin, 2011, pp. 421–432, https://doi.org/10.1007/978-3-642-23719-5_36.

[50] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea, *Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes: Finding all maximal*

α-gapped repeats and palindromes in optimal worst case time on integer alphabets, Theory Comput. Syst., 62 (2018), pp. 162–191, https://doi.org/10.1007/s00224-017-9794-5.

[51] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski, *Optimal dynamic strings*, in Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, SIAM, Philadelphia, 2018, pp. 1509–1528, https://doi.org/10.1137/1.9781611975031.99.

[52] P. Gawrychowski, M. Lewenstein, and P. K. Nicholson, *Weighted ancestors in suffix trees*, in 22nd Annual European Symposium on Algorithms, ESA 2014, LNCS 8737, Springer-Verlag, Berlin, 2014, pp. 455–466, https://doi.org/10.1007/978-3-662-44777-2_38.

[53] L. J. Guibas and A. M. Odlyzko, *Periods in strings*, J. Combin. Theory Ser. A, 30 (1981), pp. 19–42, https://doi.org/10.1016/0097-3165(81)90038-8.

[54] R. Hariharan and V. Vinay, *String matching in $\tilde{O}(\sqrt{n}+\sqrt{m})$ quantum time*, J. Discrete Algorithms, 1 (2003), pp. 103–110, https://doi.org/10.1016/S1570-8667(03)00010-8.

[55] M. C. Harrison, *Implementation of the substring test by hashing*, Commun. ACM, 14 (1971), pp. 777–779, https://doi.org/10.1145/362919.362934.

[56] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara, *Detecting regularities on grammar-compressed strings*, Inform. and Comput., 240 (2015), pp. 74–89, https://doi.org/10.1016/j.ic.2014.09.009.

[57] C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, and W. Zuba, *Linear-time computation of cyclic roots and cyclic covers of a string*, in 34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, Marne-la-Vallée, France, LIPIcs 259, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2023, pp. 15:1–15:15, https://doi.org/10.4230/LIPICS.CPM.2023.15.

[58] G. Jacobson, *Space-efficient static trees and graphs*, in 30th Annual Symposium on Foundations of Computer Science, FOCS 1989, IEEE, New York, 1989, pp. 549–554, https://doi.org/10.1109/SFCS.1989.63533.

[59] A. Jeż, *Faster fully compressed pattern matching by recompression*, ACM Trans. Algorithms, 11 (2015), pp. 20:1–20:43, https://doi.org/10.1145/2631920.

[60] A. Jeż, *Recompression: A simple and powerful technique for word equations*, J. ACM, 63 (2016), pp. 4:1–4:51, https://doi.org/10.1145/2743014.

[61] C. Jin and J. Nogler, *Quantum speed-ups for string synchronizing sets, longest common substring, and k-mismatch matching*, in 34th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, SIAM, Philadelphia, 2023, pp. 5090–5121, https://doi.org/10.1137/1.9781611977554.ch186.

[62] J. Kärkkäinen, P. Sanders, and S. Burkhardt, *Linear work suffix array construction*, J. ACM, 53 (2006), pp. 918–936, https://doi.org/10.1145/1217856.1217858.

[63] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Dev., 31 (1987), pp. 249–260, https://doi.org/10.1147/rd.312.0249.

[64] O. Keller, T. Kopelowitz, S. Landau Feibish, and M. Lewenstein, *Generalized substring compression*, Theoret. Comput. Sci., 525 (2014), pp. 42–54, https://doi.org/10.1016/j.tcs.2013.10.010.

[65] D. Kempa and T. Kociumaka, *String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure*, in 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, ACM, New York, 2019, pp. 756–767, https://doi.org/10.1145/3313276.3316368.

[66] D. Kempa and T. Kociumaka, *Resolution of the Burrows-Wheeler transform conjecture*, in 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, IEEE, New York, 2020, pp. 1002–1013, https://doi.org/10.1109/FOCS46700.2020.00097.

[67] D. Kempa and T. Kociumaka, *Dynamic suffix array with polylogarithmic queries and updates*, in 54th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2022, ACM, New York, 2022, pp. 1657–1670, https://doi.org/10.1145/3519935.3520061.

[68] D. Kempa and T. Kociumaka, *Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space*, in 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, IEEE, New York, 2023, pp. 1877–1886, https://doi.org/10.1109/FOCS57990.2023.00114.

[69] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350, https://doi.org/10.1137/0206024.

[70] T. Kociumaka, *Minimal suffix and rotation of a substring in optimal time*, in 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, LIPIcs 54, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2016, pp. 28:1–28:12, https://doi.org/10.4230/LIPIcs.CPM.2016.28.

[71] T. Kociumaka, *Efficient Data Structures for Internal Queries in Texts*, Ph.D. thesis, University of Warsaw, 2018, https://www.mimuw.edu.pl/∼kociumaka/files/phd.pdf.

[72] T. Kociumaka, R. Kundu, M. Mohamed, and S. P. Pissis, *Longest unbordered factor in quasilinear time*, in 29th International Symposium on Algorithms and Computation, ISAAC 2018, LIPIcs 123, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2018, pp. 70:1–70:13, https://doi.org/10.4230/LIPIcs.ISAAC.2018.70.

[73] T. Kociumaka, G. Navarro, and F. Olivares, *Near-optimal search time in δ-optimal space*, in 15th Latin American Symposium on Theoretical Informatics, LATIN 2022, LNCS 13568, Springer-Verlag, Berlin, 2022, pp. 88–103, https://doi.org/10.1007/978-3-031-20624-5_6.

[74] T. Kociumaka, G. Navarro, and N. Prezza, *Toward a definitive compressibility measure for repetitive sequences*, IEEE Trans. Inform. Theory, 69 (2023), pp. 2074–2092, https://doi.org/10.1109/TIT.2022.3224382.

[75] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń, *Efficient data structures for the factor periodicity problem*, in 19th International Symposium on String Processing and Information Retrieval, SPIRE 2012, LNCS 7608, Springer-Verlag, Berlin, 2012, pp. 284–294, https://doi.org/10.1007/978-3-642-34109-0_30.

[76] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń, *Internal pattern matching queries in a text and applications*, in 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, Philadelphia, 2015, pp. 532–551, https://doi.org/10.1137/1.9781611973730.36.

[77] R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov, *Searching of gapped repeats and subrepetitions in a word*, J. Discrete Algorithms, 46–47 (2017), pp. 1–15, https://doi.org/10.1016/j.jda.2017.10.004.

[78] R. M. Kolpakov and G. Kucherov, *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, IEEE, New York, 1999, pp. 596–604, https://doi.org/10.1109/SFFCS.1999.814634.

[79] T. Kopelowitz, G. Kucherov, Y. Nekrich, and T. Starikovskaya, *Cross-document pattern matching*, J. Discrete Algorithms, 24 (2014), pp. 40–47, https://doi.org/10.1016/j.jda.2013.05.002.

[80] D. Kosolobov, F. Manea, and D. Nowotka, *Detecting one-variable patterns*, in 24th International Symposium on String Processing and Information Retrieval, SPIRE 2017, LNCS 10508, Springer-Verlag, Berlin, 2017, pp. 254–270, https://doi.org/10.1007/978-3-319-67428-5_22.

[81] S. Kuruppu, S. J. Puglisi, and J. Zobel, *Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval*, in 17th International Symposium on String Processing and Information Retrieval, SPIRE 2010, LNCS 6393, Springer-Verlag, Berlin, 2010, pp. 201–206, https://doi.org/10.1007/978-3-642-16321-0_20.

[82] G. M. Landau and U. Vishkin, *Fast string matching with k differences*, J. Comput. System Sci., 37 (1988), pp. 63–78, https://doi.org/10.1016/0022-0000(88)90045-1.

[83] H. Li, *Minimap2: Pairwise alignment for nucleotide sequences*, Bioinformatics, 34 (2018), pp. 3094–3100, https://doi.org/10.1093/bioinformatics/bty191.

[84] R. C. Lyndon, *On Burnside's problem*, Trans. Amer. Math. Soc., 77 (1954), pp. 202–215, https://doi.org/10.1090/S0002-9947-1954-0064049-X.

[85] R. C. Lyndon and M.-P. Schützenberger, *The equation $a^M = b^N c^P$ in a free group*, Michigan Math. J., 9 (1962), pp. 289–298, https://doi.org/10.1307/mmj/1028998766.

[86] M. G. Main, *Detecting leftmost maximal periodicities*, Discrete Appl. Math., 25 (1989), pp. 145–153, https://doi.org/10.1016/0166-218X(89)90051-6.

[87] G. K. Manacher, *An application of pattern matching to a problem in geometrical complexity*, Inform. Process. Lett., 5 (1976), pp. 6–7, https://doi.org/10.1016/0020-0190(76)90092-2.

[88] K. Mehlhorn, R. Sundar, and C. Uhrig, *Maintaining dynamic sequences under equality tests in polylogarithmic time*, Algorithmica, 17 (1997), pp. 183–198, https://doi.org/10.1007/BF02522825.

[89] K. Mitani, T. Mieno, K. Seto, and T. Horiyama, *Internal longest palindrome queries in optimal time*, in 17th International Conference and Workshops on Algorithms and Computation, WALCOM 2023, LNCS 13973, Springer-Verlag, Berlin, 2023, pp. 127–138, https://doi.org/10.1007/978-3-031-27051-2_12.

[90] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, Cambridge, 2005, https://doi.org/10.1017/CBO9780511813603.

[91] J. H. Morris, Jr., and V. R. Pratt, *A Linear Pattern-Matching Algorithm*, Technical report 40, University of California, Berkeley, CA, 1970.

[92] J. I. Munro, Y. Nekrich, and J. S. Vitter, *Fast construction of wavelet trees*, Theoret. Comput. Sci., 638 (2016), pp. 91–97, https://doi.org/10.1016/j.tcs.2015.11.011.

[93] Y. Nekrich and G. Navarro, *Sorted range reporting*, in 13th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2012, LNCS 7357, Springer-Verlag, Berlin, 2012, pp. 271–282, https://doi.org/10.1007/978-3-642-31155-0_24.

[94] M. Patil, R. Shah, and S. V. Thankachan, *Faster range LCP queries*, in 20th International Symposium on String Processing and Information Retrieval, SPIRE 2013, LNCS 8214, Springer-Verlag, Berlin, 2013, pp. 263–270, https://doi.org/10.1007/978-3-319-02432-5_29.

[95] M. Pǎtraşcu and M. Thorup, *Dynamic integer sets with optimal rank, select, and predecessor search*, in 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, IEEE, New York, 2014, pp. 166–175, https://doi.org/10.1109/FOCS.2014.26.

[96] W. Plandowski and W. Rytter, *Application of Lempel-Ziv encodings to the solution of words equations*, in 25th International Colloquium on Automata, Languages and Programming, ICALP 1998, LNCS 1443, Springer-Verlag, Berlin, 1998, pp. 731–742, https://doi.org/10.1007/BFb0055097.

[97] M. Ponec, P. Giura, J. Wein, and H. Brönnimann, *New payload attribution methods for network forensic investigations*, ACM Trans. Inform. Syst. Secur., 13 (2010), pp. 15:1–15:32, https://doi.org/10.1145/1698750.1698755.

[98] J. Radoszewski and J. Straszyński, *Efficient computation of 2-covers of a string*, in 28th Annual European Symposium on Algorithms, ESA 2020, LIPIcs 173, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2020, pp. 77:1–77:17, https://doi.org/10.4230/LIPIcs.ESA.2020.77.

[99] E. Rivals and S. Rahmann, *Combinatorics of periods in strings*, in 28th International Colloquium on Automata, Languages and Programming, ICALP 2001, LNCS 2076, Springer-Verlag, Berlin, 2001, pp. 615–626, https://doi.org/10.1007/3-540-48224-5.

[100] E. Rivals, M. Sweering, and P. Wang, *Convergence of the number of period sets in strings*, in 50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, LIPIcs 261, Schloss Dagstuhl–Leibniz Center for Informatics, Wadern, Germany, 2023, pp. 100:1–100:14, https://doi.org/10.4230/LIPICS.ICALP.2023.100.

[101] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, *Reducing storage requirements for biological sequence comparison*, Bioinformatics, 20 (2004), pp. 3363–3369, https://doi.org/10.1093/bioinformatics/bth408.

[102] M. Ružić, *Constructing efficient dictionaries in close to sorting time*, in 35th International Colloquium on Automata, Languages and Programming, ICALP 2008, LNCS 5125, Springer-Verlag, Berlin, 2008, pp. 84–95, https://doi.org/10.1007/978-3-540-70575-8_8.

[103] S. C. Sahinalp and U. Vishkin, *On a parallel-algorithms method for string matching problems*, in 2nd Italian Conference on Algorithms and Complexity, CIAC 1994, Rome, LNCS 778, Springer-Verlag, Berlin 1994, pp. 22–32, https://doi.org/10.1007/3-540-57811-0_3.

[104] S. C. Sahinalp and U. Vishkin, *Symmetry breaking for suffix tree construction*, in 26th Annual ACM Symposium on Theory of Computing, STOC 1994, ACM, New York, 1994, pp. 300–309, https://doi.org/10.1145/195058.195164.

[105] S. C. Sahinalp and U. Vishkin, *Efficient approximate and dynamic matching of patterns using a labeling paradigm*, in 37th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1996, IEEE, New York, 1996, pp. 320–328, https://doi.org/10.1109/SFCS.1996.548491.

[106] S. Schleimer, D. S. Wilkerson, and A. Aiken, *Winnowing: Local algorithms for document fingerprinting*, in ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, ACM, New York, 2003, pp. 76–85, https://doi.org/10.1145/872757.872770.

[107] D. Sorokina, J. Gehrke, S. Warner, and P. Ginsparg, *Plagiarism detection in arXiv*, in Sixth IEEE International Conference on Data Mining, ICDM 2006, IEEE, New York, 2006, pp. 1070–1075, https://doi.org/10.1109/ICDM.2006.126.

[108] A. Thue, *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, Skrifter utgit av Videnskapsselskapet i Kristiania. I*, Matematisk-naturvidenskabelig klasse, 1 (1912), pp. 1–67, http://www.biodiversitylibrary.org/item/52278.

[109] U. Vishkin, *Deterministic sampling—A new technique for fast pattern matching*, SIAM J. Comput., 20 (1991), pp. 22–40, https://doi.org/10.1137/0220002.

[110] Q. Wang and M. Ying, *Quantum algorithm for lexicographically minimal string rotation*, Theory Comput. Syst., 68 (2024), pp. 29–74, https://doi.org/10.1007/S00224-023-10146-8.

[111] P. Weiner, *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, SWAT 1973, IEEE, New York, 1973, pp. 1–11, https://doi.org/10.1109/SWAT.1973.13.

[112] D. E. Wood and S. L. Salzberg, *Kraken: Ultrafast metagenomic sequence classification using exact alignments*, Genome Biol., 15 (2014), R46, https://doi.org/10.1186/gb-2014-15-3-r46.

[113] G. Zhou, *Two-dimensional range successor in optimal time and almost linear space*, Inform. Process. Lett., 116 (2016), pp. 171–174, https://doi.org/10.1016/j.ipl.2015.09.002.

[114] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23 (1977), pp. 337–343, https://doi.org/10.1109/TIT.1977.1055714.

[115] J. Ziv and N. Merhav, *A measure of relative entropy between individual sequences with application to universal classification*, IEEE Trans. Inform. Theory, 39 (1993), pp. 1270–1279, https://doi.org/10.1109/18.243444.