Internal Pattern Matching in Small Space and Applications

Gabriel Bathie
DIENS, École normale supérieure de Paris, PSL Research University, France
LaBRI, Université de Bordeaux, France

Panagiotis Charalampopoulos
Birkbeck, University of London, UK

Tatiana Starikovskaya
DIENS, École normale supérieure de Paris, PSL Research University, France

Abstract

In this work, we consider pattern matching variants in small space, that is, in the read-only setting, where we want to bound the space usage on top of storing the strings. Our main contribution is a space-time trade-off for the Internal Pattern Matching (IPM) problem, where the goal is to construct a data structure over a string S of length n that allows one to answer the following type of queries: Compute the occurrences of a fragment P of S inside another fragment T of S, provided that $|T| < 2|P|$. For any $\tau \in [1\,..\,n/\log^2 n]$, we present a nearly-optimal $\tilde{O}(n/\tau)$-size[1] data structure that can be built in $\tilde{O}(n)$ time using $\tilde{O}(n/\tau)$ extra space, and answers IPM queries in $O(\tau + \log n \log^3 \log n)$ time. IPM queries have been identified as a crucial primitive operation for the analysis of algorithms on strings. In particular, the complexities of several recent algorithms for approximate pattern matching are expressed with regards to the number of calls to a small set of primitive operations that include IPM queries; our data structure allows us to port these results to the small-space setting. We further showcase the applicability of our IPM data structure by using it to obtain space-time trade-offs for the longest common substring and circular pattern matching problems in the *asymmetric streaming* setting.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases internal pattern matching, longest common substring, small-space algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.26

Funding *Gabriel Bathie*: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR) and a Royal Society International Exchanges Award.
Panagiotis Charalampopoulos: Research visits during which some of the presented ideas were conceived were funded by a Royal Society International Exchanges Award.
Tatiana Starikovskaya: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR) and a Royal Society International Exchanges Award.

Acknowledgements We would like to dedicate this work to our dear friend and colleague Paweł Gawrychowski on the occasion of his 40th birthday. We thank Solon Pissis for helpful suggestions.

[1] Throughout this work, the $\tilde{O}(\cdot)$ notation suppresses factors polylogarithmic in the input-size.

1 Introduction

In the fundamental text indexing problem, the task is to preprocess a text T into a data structure (index) that can answer the following queries efficiently: Given a pattern P, find the occurrences of P in T. The Internal Pattern Matching problem (IPM) is a variant of the text indexing problem, where both the pattern P and the text T are fragments of a longer string S, given in advance.

Introduced in 2009 [47], IPM queries are a cornerstone of the family of internal queries on strings. The list of internal queries, primarily executed through IPM queries, comprises of period queries, prefix-suffix queries, periodic extension queries, and cyclic equivalence queries; see [52, 53, 50]. Other problems that have been studied in the internal setting include shortest unique substring [1], longest common substring [5], suffix rank and selection [9, 50], BWT substring compression [9], shortest absent string [10], dictionary matching [32, 21, 20], string covers [31], masked prefix sums [34], circular pattern matching [44], and longest palindrome [61].

The primary distinction between the classical and internal string queries lies in how the pattern is handled during queries. In classical queries, the input is explicitly provided at query time, whereas in internal queries, the input is specified in constant space via the endpoints of fragments of string S. This distinction enables notably faster query times in the latter setting, as there is no need to read the input when processing the query. This characteristic of IPM and similar internal string queries renders them particularly valuable for bulk processing of textual data. This is especially advantageous when S serves as input for another algorithm, as illustrated by multiple direct and indirect (via other internal queries) applications of IPM: pattern matching with variables [56, 36], detection of gapped repeats and subrepetitions [55, 41], approximate period recovery [2, 4], computing the longest unbordered substring [51], dynamic repetition detection [3], computing string covers [31], identifying two-dimensional maximal repetitions, enumeration of distinct substrings [25], dynamic longest common substring [5], approximate pattern matching [26, 27], approximate circular pattern matching [23, 24], (approximate) pattern matching with wildcards [11], RNA folding [33], and the language edit distance problem for palindromes and squares [12].

Below we assume $|T| < 2|P|$, which guarantees that the set of occurrences of P in T forms an arithmetic progression and can be thus represented in $O(1)$ space.

With no preprocessing ($O(1)$ extra space), IPM queries on a string S of length n can be answered in $O(n)$ time by a constant-space pattern matching algorithm (see [17] and references therein). On the other side of the spectrum, Kociumaka, Radoszewski, Rytter, and Waleń [52] showed that for every string $S \in [0\mathinner{.\,.}\sigma]^n$, there exists a data structure of size $O(n/\log_\sigma n)$ which answers IPM queries in optimal $O(1)$ time and can be constructed in $O(n/\log_\sigma n)$ time given the packed representation of S (meaning that S divided into blocks of $\log_\sigma n$ consecutive letters, and every block is stored in one machine word). The problem has been equally studied in the compressed and dynamic settings [26, 49, 48].

1.1 Our Main Contribution: Small-space IPM

As our main contribution, we provide a trade-off between the constant-space and $O(n)$ query time and the $O(n/\log_\sigma n)$-space and constant query time data structures. We consider the IPM problem in the read-only setting, where one assumes random read-only access to the input string(s) and only accounts for the extra space, that is, the space used by the algorithm/data structure on top of the space needed to store the input.

▶ **Corollary 1.1.** *Suppose that we have read-only random access to a n-length string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time using $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer the following internal pattern matching queries in time $O(\tau + \log n \log^3 \log n)$: given $p, p', t, t' \in [1 \mathinner{.\,.} n]$ such that $t' - t \leq 2(p' - p)$, return all occurrences of $P = S[p \mathinner{.\,.} p']$ in $T = S[t \mathinner{.\,.} t']$.*

Our data structure is nearly optimal: First, when n/τ is polynomial, the construction time is linear; and secondly, the product of the query time and space of our data structure is optimal up to polylogarithmic factors (**Lemma 3.8**).

Technical overview for IPM queries. Our solution relies heavily on utilizing the concept of τ-partitioning sets, as introduced by Kosolobov and Sivukhin [57]. For a string of length n, a τ-partitioning is a subset of $O(n/\tau)$ positions that satisfies some density and consistency criteria. We use the positions of such a set as anchor points for identifying pattern occurrences, provided that the pattern avoids a specific periodic structure. To detect these anchored occurrences, we employ sparse suffix trees alongside a three-dimensional range searching structure. In cases where the pattern does not avoid said periodic structure, we employ a different strategy, leveraging the periodic structure to construct the necessary anchor points.

We next provide a brief comparison of the outlined approach with previous work. String anchoring techniques have been proven very useful in and been developed for text indexing problems, such as the longest common extension (LCE) problem, in small space [57, 16]. One of the most technically similar works to ours is that of Ben-Nun et al. [14] who considered the problem of computing a long common substring of two input strings in small space. They use an earlier variant of τ-partitioning sets, due to Birenzwige et al. [16], that has slightly worse guarantees than that of Kosolobov and Shivukhin [57]. The construction of anchors for substrings with periodic structure is quite similar to that of Ben-Nun et al. [14]. After computing a set of anchors, they aim to identify a synchronised pair of anchors that yields a long common substring; they achieve this via mergeable AVL trees. As IPM queries need to be answered in an online manner, we instead construct an appropriate orthogonal range searching data structure over a set of points that correspond to anchors. Using orthogonal range searching is a by-now classical approach for text indexing, see [58] for a survey.

1.2 Applications

Several internal queries reduce to IPM queries, and hence we obtain efficient implementations of them in the small-space setting. Additionally, we port several efficient approximate pattern matching algorithms to the small-space setting since IPM was the only primitive operation that they rely on that did not have an efficient small-space implementation to this day. See Section 4 for details on these applications.

Longest Common Substring (LCS). The LCS problem is formally defined as follows.

LONGEST COMMON SUBSTRING (LCS)
Input: Strings S and T of length at most n.
Output: The length of a longest string that appears as a (contiguous) fragment in both S and T.

The length of the longest common substring is one of the most popular string-similarity measures. The by-now classical approach to the LCS problem is to construct the suffix tree of S and T in $O(n)$ time and space. The longest common substring of the two strings appears as a common prefix of a pair of suffixes of S and T and hence its length is the maximal string-depth of a node of the suffix tree with leaf-descendants corresponding to suffixes of both strings; this node can be found in $O(n)$ time in a bottom-up manner.

Starikovskaya and Vildhøj [64] were the first to consider the problem in the read-only setting. They showed that for any $n^{2/3} < \tau \le n$, the problem can be solved in $O(\tau)$ extra space and $O(n^2/\tau)$ time. Kociumaka et al. [54] extended their bound to all $1 \le \tau \le n$, which in particular resulted in a constant-space read-only algorithm running in time $\tilde{O}(n^2)$.

In an attempt to develop even more space-efficient algorithms for the LCS problem, it might be tempting to consider the streaming setting, which is particularly restrictive: in this setting, one assumes that the input arrives letter-by-letter, as a stream, and must account for all the space used. Unfortunately, this setting does not allow for better space complexity: any streaming algorithm for LCS, even randomised, requires $\Omega(n)$ bits of space (**Theorem 5.2**). In the asymmetric streaming setting, which is slightly less restrictive and was introduced by Andoni et al. [7] and Saks and Seshadhri [63], the algorithm has random access to one string and sequential access to the other. Mai et al. [60] showed that in this setting, LCS can be solved in $\tilde{O}(n^2)$ time and $O(1)$ space. By utilising (a slightly more general variant of) IPM queries, we extend their result and show that for every $\tau \in [\sqrt{n}\log n(\log\log n)^3 \mathinner{.\,.} n]$, there is an asymmetric streaming algorithm that solves the LCS problem in $O(\tau)$ space and $\tilde{O}(n^2/\tau)$ time (**Theorem 6.1**). Note that these bounds almost match the bounds of Kociumaka et al. [54], while the setting is stronger.

Circular Pattern Matching (CPM). The CPM problem is formally defined as follows.

CIRCULAR PATTERN MATCHING (CPM)
Input: A pattern P of length m, a text T of length n.
Output: All occurrences of rotations of P in T.

The interest in occurrences of rotations of a given pattern is motivated by applications in Bioinformatics and Image Processing: in Bioinformatics, the starting position of a biological sequence can vary significantly due to the arbitrary nature of sequencing in circular molecular structures or inconsistencies arising from different standards of linearization applied to sequence databases; and in Image Processing, the contour of a shape can be represented using a directional chain code, which can be viewed as a circular sequence, particularly when the orientation of the image is irrelevant [8].

For strings over an alphabet of size σ, the classical read-only solution for CPM via the suffix automaton of $P \cdot P$ runs in $O(n \log \sigma)$ time and uses $O(m)$ extra space [59]. Recently, Charalampopoulos et al. showed a simple $O(n)$ time and $O(m)$ extra space solution. The problem has been also studied from the practical point of view [65, 40, 29] and in the text indexing setting [45, 43, 42].

It is not hard to see that the CPM and the LCS problems are closely related: occurrences of rotations of P in T are exactly the common substrings of $P \cdot P$ and T of length m. Implicitly using this observation, we show that in the streaming setting, the CPM problem requires $\Omega(m)$ bits of space (**Theorem 5.3**) and that in the asymmetric streaming setting, for every $\tau \in [\sqrt{m}\log m(\log\log m)^3 \mathinner{.\,.} m]$, there exists an algorithm that solves the CPM problem in time $\tilde{O}(mn/\tau)$ using $O(\tau)$ extra space (**Corollary 6.5**). Finally, in the read-only setting, we give an *online* $O(n)$-time, $O(1)$-space algorithm (**Theorem 7.1**).

2 Preliminaries

For integers $i, j \in \mathbb{Z}$, denote $[i \mathinner{.\,.} j] = \{k \in \mathbb{Z} : i \le k \le j\}$, $[i \mathinner{.\,.} j) = \{k \in \mathbb{Z} : i \le k < j\}$. We consider an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ of size polynomially bounded in the length of the input string(s). The elements of the alphabet are called letters, and a string is a finite sequence of letters. For a string T and an index $i \in [1\mathinner{.\,.} n]$, the i-th letter of T is denoted by $T[i]$. We use $|T| = n$ to denote the length of T. For two strings S, T, we use ST or $S \circ T$ indifferently to denote their concatenation $S[1] \cdots S[|S|]T[1] \cdots T[|T|]$. For integers i, j, $T[i\mathinner{.\,.} j]$ denotes the *fragment* $T[i]T[i+1] \cdots T[j]$ of T if $1 \le i \le j \le n$ and the empty string ε otherwise. We extend this notation in a natural way to $T[i\mathinner{.\,.} j+1) = T[i\mathinner{.\,.} j] = T(i-1\mathinner{.\,.} j]$. When $i = 1$ or $j = n$, we omit these indices, i.e., $T[\mathinner{.\,.} j] = T[1\mathinner{.\,.} j]$ and $T[i\mathinner{.\,.}] = T[i\mathinner{.\,.} n]$. A string P is a *prefix* of T if there exists $j \in [1\mathinner{.\,.} n]$ such that $P = T[\mathinner{.\,.} j]$, and a *suffix* of T if there exists $i \in [1\mathinner{.\,.} n]$ such that $P = T[i\mathinner{.\,.}]$. We denote the reverse of a string T by $\mathrm{rev}(T) = T[n]T[n-1] \cdots T[2]T[1]$. For an integer $\Delta \in [1\mathinner{.\,.} n]$, we say that a string $T[\Delta+1\mathinner{.\,.} n] \circ T[1\mathinner{.\,.} \Delta]$ is a *rotation* of T. A fragment $T[i\mathinner{.\,.} j]$ of a string T is called an *occurrence* of a string P if $T[i\mathinner{.\,.} j] = P$; in this case, we say that P *occurs* at position i of T. A positive integer ρ is a *period* of a string T if $T[i] = T[i+\rho]$ for all $i \in [1\mathinner{.\,.} |T| - \rho]$. The smallest period of T is referred to as *the period* of T and is denoted by $\mathsf{per}(T)$. If $\mathsf{per}(T) \le |T|/2$, T is called *periodic*.

▶ **Fact 2.1** (Corollary of the Fine–Wilf periodicity lemma [37]). *The starting positions of the occurrences of a pattern P in a text T form $O(|T|/|P|)$ arithmetic progressions with difference $\mathsf{per}(P)$.*

We assume a reader to be familiar with basic data structures for string processing, see, e.g., [59]. Recall that a suffix tree for a string S is essentially a compact trie representing the set of all suffixes of S, whereas a sparse suffix tree contains only a subset of these suffixes.

▶ **Fact 2.2** ([57, Theorem 3]). *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $b = \Omega(\log^2 n)$, one can construct in $O(n \log_b n)$ time and $O(b)$ space the sparse suffix tree for arbitrarily chosen b suffixes.*

▶ **Fact 2.3** ([17]). *There is a read-only online algorithm that finds all occurrences of a pattern P of length m in a text T of length $n \ge m$ in $O(n)$ time and $O(1)$ space.*

▶ **Fact 2.4** ([38, Lemma 6]). *Given read-only random access to a string S of length n, one can decide in $O(n)$ time and $O(1)$ space if S is periodic and, if so, compute $\mathsf{per}(S)$.*

▶ **Fact 2.5** ([35]). *Given read-only random access to a string S of length n, the lexicographically smallest rotation of a string S can be computed in $O(n)$ time and $O(1)$ space.*

Static predecessor. For a static set, a combination of x-fast tries [66] and deterministic dictionaries [62] yields the following efficient deterministic data structure; cf. [39].

▶ **Fact 2.6** ([39, Proposition 2]). *A sorted static set $Y \subseteq [1\mathinner{.\,.} U]$ can be preprocessed in $O(|Y|)$ time and space so that predecessor queries can be performed in $O(\log\log |U|)$ time.*

Weighted ancestor queries. Let $\mathcal{T}$ be a rooted tree with integer weights on nodes. A *weighted ancestor* query for a node u and weight d must return the highest ancestor of u with weight at least d.

▶ **Fact 2.7** ([6]). *Let $\mathcal{T}$ be a rooted tree of size n with integer weights on nodes. Assume that each weight is at most n, with the weight of the root being zero, and the weight of every non-root node being strictly larger than its parent's weight. $\mathcal{T}$ can be preprocessed in $O(n)$ time and space so that weighted ancestor queries on it can be performed in $O(\log\log n)$ time.*

If $\mathcal{T}$ is the suffix tree of a string and the weights are the string-depths of the nodes, this result can be improved further:

▶ **Fact 2.8** ([13]). *The suffix tree $\mathcal{T}$ of a string of length n can be preprocessed in $O(n)$ time and $O(n)$ space so that weighted ancestor queries on it can be performed in $O(1)$ time.*

3D range emptiness. A three-dimensional orthogonal range emptiness query asks whether a range $[a_1 \times a_2] \times [b_1 \times b_2] \times [c_1 \times c_2]$ is empty.

▶ **Fact 2.9** ([46, Theorem 2]). *There exists a data structure that answers three-dimensional orthogonal range emptiness queries on a set of n points from a $[U] \times [U] \times [U]$ grid in $O(\log\log U + (\log\log n)^3)$ time, uses $O(n \log n (\log\log n)^3)$ space, and can be constructed in $O(n \log^4 n \log\log n)$ time. If the query range is not empty, the data structure also outputs a point from it.*

▶ Remark 2.10. Better space vs. query-time tradeoffs than the above are known for the 3D range emptiness problem; cf [19] and references therein. We opted for the data structure encapsulated of Fact 2.9 due to its efficient construction algorithm. Note that a data structure capable of reporting all points in an orthogonal range over a $[U] \times [U] \times [U]$ grid with n points in time $O(Q_1(U, n) + Q_2(U, n) \cdot |\mathsf{output}|)$ can answer range emptiness queries, also returning a witness in the case the range is not empty, in time $O(Q_1(U, n) + Q_2(U, n))$.

3 Internal Pattern Matching

We consider a slightly more powerful variant of IPM queries, as required by our applications. A reader that is only interested in IPM queries can focus on the case when $a = \varepsilon$.

Extended IPM (Decision)
Input: A string S of length n over an integer alphabet to which we have read-only random access.
Query: Given $p, p', t, t' \in [1 \,..\, n]$ and $a \in \Sigma \cup \{\varepsilon\}$, return whether $P := S[p \,..\, p']a$ occurs in $T := S[t \,..\, t']$ and, if so, return a witness occurrence.

Our solution for Extended IPM (Decision) heavily relies on a solution for the following auxiliary problem.

Anchored IPM
Input: A string S of length n over an integer alphabet Σ to which we have read-only random access and a set $\mathcal{A} \subseteq [1 \,..\, n]$.
Query: Given $p, x, p', t, t' \in [1 \,..\, n]$ with $p \le x \le p'$, $x \in \mathcal{A}$, and $a \in \Sigma \cup \{\varepsilon\}$, for $P := S[p \,..\, p']a$, decide whether there exists an occurrence of P at some position $j \in [t \,..\, t' - |P| + 1]$ such that $j + (x - p) \in \mathcal{A}$ and, if so, return a witness.

▶ **Lemma 3.1.** *There exists a data structure for the Anchored IPM problem that can be built using $O(n \log_{|\mathcal{A}|} n) + O(|\mathcal{A}| \log^4 |\mathcal{A}| \log\log |\mathcal{A}|)$ time and $O(|\mathcal{A}| \log |\mathcal{A}| (\log\log |\mathcal{A}|)^3)$ extra space, and answers queries in $O(\log^3 \log n)$ time.*

Proof. For an integer $y \in [1 \,..\, n]$, denote $P_y := \mathrm{rev}(S[\,..\, y])$ and $S_y := S[y \,..\,]$. Consider a family $\mathcal{X} := \{(P_y\$, S_y\$) : y \in \mathcal{A}\}$ of pairs of strings, where $\$ \notin \Sigma$ is a letter lexicographically smaller than all others. Using Fact 2.2, we build a sparse suffix tree RSST for the first components of the elements of $\mathcal{X}$ and a sparse suffix tree SST for the second components of the elements of $\mathcal{X}$.

Consider a three-dimensional grid $[1\,.\,.\,n] \times [1\,.\,.\,n] \times [1\,.\,.\,n]$. In this grid, create a set Π of points, which contains, for each element $(P_y\$, S_y\$)$ of $\mathcal{X}$, a point $(\mathsf{rank}_{\text{rev}}(y), \mathsf{rank}(y), y)$, where $\mathsf{rank}_{\text{rev}}(y)$ is the lexicographic rank of $P_y\$$ among the first components of the elements of $\mathcal{X}$ and $\mathsf{rank}(y)$ is the lexicographic rank of $S_y\$$ among the second components of the elements of $\mathcal{X}$.

Upon a query, we first retrieve the leaves corresponding to $P_x\$$ and $S_x\$$ in RSST and SST, respectively. This can be done in $O(\log\log n)$ time with the aid of Fact 2.6 built over the elements of $\mathcal{A}$, with $x \in \mathcal{A}$ storing pointers to the corresponding leaves as satellite information. Next, we retrieve the (possibly implicit) nodes u and v corresponding to $\text{rev}(S[p\,.\,.\,x))$ in RSST and $S[x\,.\,.\,p']a$ in SST, respectively. This can be done in $O(\log\log n)$ time after an $O(|\mathcal{A}|)$-time preprocessing of (a) the two trees according to Fact 2.7 and (b) the edge-labels of the outgoing edges of each node using Fact 2.6. Now, it suffices to check if there is some integer j such that the leaf corresponding to $P_j\$$ is a descendant of u, the leaf corresponding to $S_j\$$ is a descendant of v, and $j \in [t + (x - p)\,.\,.\,t' - (p' + |a| - x)]$. After a linear-time bottom-up preprocessing of RSST and SST, we can retrieve in $O(1)$ time the following ranges:

- $R_1 = \{\mathsf{rank}_{\text{rev}}(y) :$ the node of RSST corresponding to $P_y\$$ is a descendant of $u\}$;
- $R_2 = \{\mathsf{rank}(y) :$ the node of SST corresponding to $S_y\$$ is a descendant of $v\}$.

The query then reduces to deciding whether the orthogonal range $R_1 \times R_2 \times [t + (x-p)\,.\,.\,t' - (p' + |a| - x)]$ contains any point in Π, and returning a witness if it does. We can do this efficiently by building the data structure encapsulated in Fact 2.9 for Π: the query time is $O(\log^3 \log n)$, while the construction time is $O(n \log_{|\mathcal{A}|} n) + O(|\mathcal{A}| \log^4 |\mathcal{A}| \log\log |\mathcal{A}|)$ and the space usage is $O(|\mathcal{A}| \log |\mathcal{A}| (\log\log |\mathcal{A}|)^3)$. ◀

For an integer parameter τ, we next present a data structure for Extended IPM (Decision) that uses $\tilde{O}(n/\tau)$ space on top of the space required to store S and answers queries in nearly-constant time provided that P is of length greater than 5τ. We achieve this result using the so-called τ-partitioning sets of Kosolobov and Sivukhin [57] as *anchors* for the occurrences if P avoids a certain periodic structure, and by exploiting said periodic structure to construct anchors in the remaining case.

▶ **Definition 3.2** (τ-partitioning set). *Given an integer $\tau \in [4\,.\,.\,n/2]$, a set of positions $\mathcal{P} \subseteq [1\,.\,.\,n]$ is called a* τ-partitioning set *if it satisfies the following properties:*

(a) *if $S[i-\tau\,.\,.\,i+\tau] = S[j-\tau\,.\,.\,j+\tau]$ for $i, j \in [\tau+1\,.\,.\,n-\tau]$, then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$;*

(b) *if $S[i\,.\,.\,i+\ell] = S[j\,.\,.\,j+\ell]$, for $i, j \in \mathcal{P}$ and some $\ell \geq 0$, then, for each $d \in [0\,.\,.\,\ell-\tau)$, $i + d \in \mathcal{P}$ if and only if $j + d \in \mathcal{P}$;*

(c) *if $i, j \in [1\,.\,.\,n]$ with $j - i > \tau$ and $(i\,.\,.\,j) \cap \mathcal{P} = \emptyset$, then $S[i\,.\,.\,j]$ has period at most $\tau/4$.*

▶ **Theorem 3.3** ([57]). *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4\,.\,.\,O(n/\log^2 n)]$ and $b = n/\tau$, one can construct in $O(n \log_b n)$ time and $O(b)$ extra space a τ-partitioning set $\mathcal{P}$ of size $O(b)$. The set $\mathcal{P}$ additionally satisfies the property that if a fragment $S[i\,.\,.\,j]$ has period at most $\tau/4$, then $\mathcal{P} \cap [i + \tau\,.\,.\,j - \tau] = \emptyset$.*

▶ **Definition 3.4** (τ-runs). *A fragment F of a string S is a τ-run if and only if $|F| > 3\tau$, $\mathsf{per}(F) \leq \tau/4$, and F cannot be extended in either direction without its period changing. The Lyndon root of a τ-run R is the lexicographically smallest rotation of $R[1\,.\,.\,\mathsf{per}(R)]$.*

The following fact follows from the proof of Lemma 10 in the full version of [22], where the definition of τ-runs is slightly different, but captures all of our τ-runs.

▶ **Fact 3.5** (cf. [22, proof of Lemma 10]). *Two τ-runs can overlap by at most $\tau/2$ positions. The number of τ-runs in a string of length n is $O(n/\tau)$.*

▶ **Lemma 3.6.** *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4\mathinner{.\,.} O(n/\log^2 n)]$, all τ-runs in S can be computed and grouped by Lyndon root in $O(n \log_b n)$ time using $O(b)$ extra space, where $b = n/\tau$. Within the same complexities, we can compute, for each τ-run, the first occurrence of its Lyndon root in it.*

Proof. We first compute a τ-partitioning set $\mathcal{P}$ for S using Theorem 3.3. Due to Property c, its converse that is stated in Theorem 3.3, and Fact 3.5 there is a natural injection from the τ-runs to the maximal fragments of length at least τ that do not contain any position in $\mathcal{P}$ — the τ-run corresponding to such a maximal fragment may extend for τ more positions in each direction. We can find the period of each maximal fragment in time proportional to its length using $O(1)$ extra space due to Fact 2.4. We then try to extend the maximal fragment to a τ-run using $O(\tau)$ letter comparisons. Additionally, we compute the Lyndon root of each computed τ-run R in $O(\tau) = O(|R|)$ time by applying Fact 2.5 to $R[1\mathinner{.\,.}\mathsf{per}(R)]$. The first occurrence of the Lyndon root in the τ-run can be computed in constant time since we know which rotation of $R[1\mathinner{.\,.}\mathsf{per}(R)]$ equals the Lyndon root. Over all τ-runs, the total time is $O(n)$ due to Fact 3.5. ◀

We next prove the main result of this section.

▶ **Theorem 3.7.** *For any $\ell \in [20\mathinner{.\,.} O(n/\log^2 n)]$, there is a data structure for* Extended IPM *(*Decision*) that can be built using $O(n \log_{n/\ell} n) + O((n/\ell) \cdot \log^4 n \log\log n)$ time and $O((n/\ell) \cdot \log n (\log\log n)^3)$ extra space given random access to S and answers queries in $O(\log^3 \log n)$ time, provided that $|P| > \ell$.*

Proof. Let $\tau = \lfloor \ell/5 \rfloor$. We use Theorem 3.3 and Lemma 3.6 with parameter τ to compute a partitioning set $\mathcal{P}$ of size $O(n/\tau)$ and all τ-runs in S, grouped by Lyndon root, each one together with the first occurrence of its Lyndon root. We create a static predecessor structure $\mathcal{R}$ using Fact 2.6, where we insert the starting position of each run R with the following satellite information: R's ending position, the first occurrence of R's Lyndon root in R, and an identifier of its group. We additionally create a data structure $\mathcal{Q}$, where, for each group of τ-runs with a common root L, indexed by their identifiers, we construct, using Fact 2.6, a predecessor data structure for a set $Q_L := \{(y, s, e) : S[s\mathinner{.\,.} e] \text{ is the longest } \tau\text{-run with a suffix } L \circ L[1\mathinner{.\,.} y]\}$, with the first components being the keys and the remaining components being stored as satellite information. The sets Q_L can be straightforwardly constructed in $O(n \log n/\tau)$ time.

Now, let $\mathcal{L}$ be a set that contains the ending position of each τ-run as well as the starting (resp. ending) positions of the first (resp. last) two occurrences of the Lyndon root in this τ-run; $\mathcal{L}$ can be straightforwardly constructed in $O(n/\tau)$ time given the information returned by the application of Lemma 3.6. We then construct a set $\mathcal{A} := \mathcal{P} \cup \mathcal{L}$ and preprocess the string S and the set $\mathcal{A}$ according to Lemma 3.1.

Our query comprises of two steps.

Step 1: First, we deal with the case when both P and T have period at most $\tau/4$. Since P and T are of length at least 5τ, due to Fact 3.5, each of them can be only contained in the τ-run whose starting position is closest to it in the left. We can thus check whether they both have period at most $\tau/4$ in $O(\log\log n)$ time by performing two predecessor queries on $\mathcal{R}$. If this turns out to be the case, we then check whether the two corresponding τ-runs belong to the same group. If they do not, then P does not occur in T due to Fact 3.5. Otherwise, let the common Lyndon root of the two runs be L. We can compute in constant time non-negative integers $x_P, x_T, y_P, y_T < |L|$ and e_P, e_T such that $P = L(|L| - x_P\mathinner{.\,.}] \circ L^{e_P} \circ L[\mathinner{.\,.} y_P]$ and $T = L(|L| - x_T\mathinner{.\,.}] \circ L^{e_T} \circ L[\mathinner{.\,.} y_T]$. Note that P occurs in T if and only if at least one of the following conditions is met: (1) $e_P = e_T$, $x_P \leq x_T$, and $y_P \leq y_T$; or (2) $e_P = e_T - 1$ and $x_P \leq x_T$; or (3) $e_P = e_T - 1$ and $y_P \leq y_T$; or (4) $e_P \leq e_T - 2$. In each of the four cases, we can compute an occurrence of P in T in constant time.

Step 2: In the second step of the query, we consider the case when $\mathsf{per}(T) > \tau/4$ and distinguish between two cases depending on whether $\mathsf{per}(S[p\mathinner{.\,.} p+3\tau]) \leq \tau/4$. In each case, it suffices to perform at most two anchored internal pattern matching queries.

Case I: $\mathsf{per}(S[p\mathinner{.\,.} p+3\tau]) > \tau/4$. Due to Property c, $[p\mathinner{.\,.} p+3\tau] \cap \mathcal{P} \neq \emptyset$. Let $x = \min([p\mathinner{.\,.} p+3\tau] \cap \mathcal{P})$. Additionally, due to Property b, for any occurrence of P in S at position j, we have $[p\mathinner{.\,.} p+3\tau] \cap \mathcal{P} = (p-j) + ([j\mathinner{.\,.} j+3\tau] \cap \mathcal{P})$, and hence $j + (x-p) \in \mathcal{P}$. Thus, an anchored IPM query returns the desired answer in $O(\log^3 \log n)$ time.

Case II: $\mathsf{per}(S[p\mathinner{.\,.} p+3\tau]) \leq \tau/4$. We distinguish between two subcases depending on whether $\mathsf{per}(P) > \tau/4$; we can check this in $O(\log\log n)$ time with the aid of data structure $\mathcal{R}$ by comparing p' with the ending position of the τ-run that contains $S[p\mathinner{.\,.} p+3\tau]$ and checking if $a = P[|P| - \mathsf{per}(S[p\mathinner{.\,.} p+3\tau])]$ if $a \neq \varepsilon$.

Subcase (a): $\mathsf{per}(P) > \tau/4$. In this case, for any occurrence of P in T, the ending position of the τ-run that is a prefix of P must be aligned with the ending position of a τ-run in T, which belongs to $\mathcal{L} \subseteq \mathcal{A}$.

Recall that $P = S[p\mathinner{.\,.} p']a$. If the period of $S[p\mathinner{.\,.} p']$ is greater than $\tau/4$, we retrieve the ending position of the τ-run containing $S[p\mathinner{.\,.} p+3\tau]$, which is in $\mathcal{L} \subseteq \mathcal{A}$ as well and issue an anchored internal pattern matching query. Assume now that the period of $S[p\mathinner{.\,.} p'+1]$ is at most $\tau/4$ and $\varepsilon \neq a \neq P[|P| - \mathsf{per}(S[p\mathinner{.\,.} p'])]$, in which case p' might not be in $\mathcal{A}$. In this case, we retrieve a fragment $S[q\mathinner{.\,.} q']$ equal to $S[p\mathinner{.\,.} p']$, such that q' is an ending position of a τ-run in $O(\log\log n)$ time using the data structure $\mathcal{Q}$, if such a fragment exists, and use $q \in \mathcal{L} \subseteq \mathcal{A}$ as the anchor to our internal anchor query, effectively searching for $S[q\mathinner{.\,.} q']a = P$. Observe that if such a fragment $S[q\mathinner{.\,.} q']$ does not exist, P cannot have any occurrence in T.

Subcase (b): $\mathsf{per}(P) \leq \tau/4$. We consider an occurrence of P in the τ-run that contains P that starts in its first $\mathsf{per}(P)$ positions and one that ends in its last $\mathsf{per}(P)$ positions. Let these two occurrences be at positions p_1 and p_2, respectively. Each of these occurrences contains at least one element of $\mathcal{L}$; let those elements be denoted q_1 for the occurrence at p_1 and q_2 for the occurrence at p_2.

Note that these elements can be straightforwardly computed given the endpoints of the τ-run, the endpoints of P, and the first occurrence of the Lyndon root in the τ-run, which we already have in hand. We then issue anchored internal pattern matching queries for $(p_1, q_1, p_1 + |P| - 1, t, t')$ and $(p_2, q_2, p_2 + |P| - 1, t, t')$ as both q_1 and q_2 are in $\mathcal{L}$. These queries are answered in $O(\log^3 \log n)$ time. As we show next, if P has an occurrence in T, this occurrence will be returned by those queries.

Consider an occurrence of P in $S[t\mathinner{.\,.} t']$ and denote the τ-run that contains this occurrence by R. Since $\mathsf{per}(T) > \tau/4$, R does not contain $S[t\mathinner{.\,.} t']$. Without loss of generality, let us assume that R does not extend to the left of $S[t\mathinner{.\,.} t']$, the remaining case is symmetric. Let the first occurrence of the Lyndon root L of the τ-run in P be at position $i = q_1 - p_1 + 1$ of P, noting that $i \leq \mathsf{per}(P)$. Then, in the leftmost occurrence of P in R, position i must be aligned with either the first or the second position where L occurs in R. By the construction of the set $\mathcal{L}$, it follows that both of these positions are in $\mathcal{L}$, and hence the anchored internal pattern matching query will return an occurrence. ◀

▶ **Corollary 1.1.** *Suppose that we have read-only random access to a n-length string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time using $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer the following internal pattern matching queries* *in time* $O(\tau + \log n \log^3 \log n)$*: given* $p, p', t, t' \in [1 \,.\,.\, n]$ *such that* $t' - t \leq 2(p' - p)$*, return all occurrences of* $P = S[p \,.\,.\, p']$ *in* $T = S[t \,.\,.\, t']$*.*

Proof. If the length of P is at most $\max\{\tau, 20\}$, we compute its occurrences in T, whose length is $O(\tau)$, in $O(\tau)$ time using Fact 2.3. In what follows, we assume that $|P| > \max\{\tau, 20\}$.

We build the EXTENDED IPM (DECISION) data structure of Theorem 3.7 for S with $\ell = \max\{\tau, 20\}$. This allows us to efficiently answer the decision version of the desired IPM queries, also returning a witness, in $O(\log^3 \log n)$ time. If the query does not return an occurrence of P in T, we are done. Otherwise, we have to compute all occurrences of P in T represented as an arithmetic progression (cf Fact 2.1). Let the witness returned by the data structure be $S[x \,.\,.\, x']$. Consider the rightmost occurrence of P in $S[t \,.\,.\, x')$, or, if it does not exist, the leftmost occurrence in $S(x \,.\,.\, t']$. Such an occurrence can be found by binary search. If no such occurrence exists, we are again done, as P has a single occurrence in T. Otherwise, the occurrences of P in T form an arithmetic progression with difference equal to the difference d of x and the starting position of the found occurrence due to Fact 2.1. We compute the extreme values of this arithmetic progression using binary search as well: we compute the minimum and the maximum $j \in \mathbb{Z}$ such that $S[x + j \cdot d \,.\,.\, x' + j \cdot d] = P$ and $t \leq x + j \cdot d \leq x' + j \cdot d \leq t'$ using $O(\log n)$ IPM queries in total; the complexity follows. ◀

3.1 Lower Bound for an IPM data structure

We now show that the product of the query time and the space achieved in Corollary 1.1 is optimal up to polylogarithmic factors, via a reduction from the following problem.

LONGEST COMMON EXTENSION (LCE)
Input: A string S of length n.
Query: Given $i, j \in [1 \,.\,.\, n]$, return the largest ℓ such that $S[i \,.\,.\, i + \ell] = S[j \,.\,.\, j + \ell]$.

Bille et al. [15, Lemma 4] showed that any data structure for LCE for n-length strings that uses s bits of extra space on top of the input has query time $\Omega(n/s)$.

▶ **Lemma 3.8.** *In the non-uniform cell-probe model, any IPM data structure that uses* s *bits of space on top of the input for a string of length* n*, has query time* $\Omega(n/(s \log n))$*.*

Proof. We prove Lemma 3.8 by reducing LCE queries in a string S of length n to IPM queries in S. Consider an IPM data structure with space s and query time q and observe that IPM queries can be used to check substring equality since $S[i \,.\,.\, i'] = S[j \,.\,.\, j']$ if and only if $S[i \,.\,.\, i']$ occurs inside the interval $[j \,.\,.\, j']$ and $j' - j = i' - i$. Using binary search, we can thus answer any LCE query via $O(\log n)$ IPM queries. Hence, we have $q \log n = \Omega(n/s)$, which concludes the proof the lemma. ◀

Lemma 3.8 implies a similar lower bound for the word RAM model, which is weaker than the non-uniform cell-probe model.

4 Other Internal Queries and Approximate Pattern Matching

In the `PILLAR` model of computation [26] the runtimes of algorithms are analysed with respect to the number of calls made to standard word-RAM operations and a few primitive string operations. It has been used to design algorithms for internal queries [52, 53, 50], approximate pattern matching under Hamming distance [26] and edit distance [27], circular approximate pattern matching under Hamming distance [24] and edit distance [28], and (approximate) wildcard pattern matching under Hamming distance [11]. Space-efficient implementations of the PILLAR model immediately result in space-efficient implementations of the above algorithms.

In the PILLAR model, one is given a family of strings $\mathcal{X}$ for preprocessing. The elementary objects are fragments $X[i\mathinner{.\,.} j]$ of strings $X \in \mathcal{X}$. Each fragment S is represented via a handle, which is how S is passed as input to PILLAR operations. Initially, the model provides a handle to each $X \in \mathcal{X}$. Handles to other fragments can be obtained through an Extract operation:

- $\mathsf{Extract}(S, \ell, r)$: Given a fragment S and positions $1 \le \ell \le r \le |S|$, extract $S[\ell\mathinner{.\,.} r]$.

Furthermore, given elementary objects S, S_1, S_2 the following primitive operations are supported in the PILLAR model:

- $\mathsf{Access}(S, i)$: Assuming $i \in [1\mathinner{.\,.} |S|]$, retrieve $S[i]$.
- $\mathsf{Length}(S)$: Retrieve the length $|S|$ of S.
- Longest common prefix $\mathsf{LCE}(S_1, S_2)$: Compute the length of the longest common prefix of S_1 and S_2.
- $\mathsf{LCE}^R(S_1, S_2)$: Compute the length of the longest common suffix of S_1 and S_2.
- Internal pattern matching $\mathsf{IPM}(S_1, S_2)$: Assuming that $|S_2| < 2|S_1|$, compute the set of the starting positions of occurrences of S_1 in S_2 represented as one arithmetic progression.

All PILLAR operations other than LCE, LCE^R, and IPM admit trivial constant-time and constant-space implementations in the read-only setting. For any $\tau = O(n/\log^2 n)$, Kosolobov and Sivukhin [57] showed that after $O(n \log_{n/\tau} n)$-time, $O(n/\tau)$-space preprocessing, LCE and LCE^R queries can be supported in $O(\tau)$ time. For IPM queries, we use Corollary 1.1.

In [52, 53, 50] it is (implicitly) shown that the following internal queries can be efficiently implemented in the PILLAR model.

- A *cyclic equivalence query* takes as input two equal-length fragments $U = S[i\mathinner{.\,.} i+\ell]$ and $V = S[j\mathinner{.\,.} j+\ell]$, and returns all rotations of U that are equal to V. Any cyclic equivalence query reduces to $O(1)$ LCE queries and $O(1)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = O(1)$.
- A *period query* takes as input a fragment $U = S[i\mathinner{.\,.} j]$, and returns all periods of U. Such a period query reduces to $O(\log |U|)$ LCE queries and $O(\log |U|)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = O(1)$.
- A *2-period* query takes as input a fragment $U = S[i\mathinner{.\,.} j]$, checks if U is periodic and, if so, it also returns U's period. Such a query reduces to $O(1)$ LCE queries and $O(1)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = O(1)$.

▶ **Corollary 4.1.** *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time and $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer cyclic equivalence and 2-period queries on S in $O(\tau + \log n \log^3 \log n)$ time, and period queries on S in $O(\tau \log n + \log^2 n \log^3 \log n)$ time.*

By plugging this implementation of the PILLAR model into [26, 27, 24, 11, 28], we obtain the following:

▶ **Corollary 4.2.** *Suppose that we have read-only random access to a text T of length n, a pattern P of length m over an integer alphabet. Given an integer threshold k, for any integer $\tau = O(m/\log^2 m)$, we can compute:*

- *the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n + k^2\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^2)$ extra space;*
- *the approximate occurrences of P in T under the edit distance in $\tilde{O}(n + k^{3.5}\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^{3.5})$ extra space;*

- *the approximate occurrences of all rotations of P in T under the Hamming distance in $\tilde{O}(n + k^3\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^3)$ extra space;*
- *the approximate occurrences of all rotations of P in T under the edit distance in $\tilde{O}(n + k^5\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^5)$ extra space;*
- *in the case where P has D wildcard letters arranged in G maximal intervals, the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n + (D+k)(G+k)\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + (D+k)(G+k))$ extra space.*

To the best of our knowledge, the only work that has considered approximate pattern matching in the read-only model is due to Bathie et al. [12]. They presented online algorithms both for the Hamming distance and the edit distance; for the Hamming distance their algorithm uses $O(k \log m)$ extra space and $O(k \log m)$ time per letter of the text, and for the edit distance $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortised time per letter.

5 LCS and CPM in the Streaming Setting

In the streaming setting, we receive a stream composed of the concatenation of the input strings, e.g., the pattern and the text in the case of CPM. We account for all the space used, including the space needed to store any information about the input strings. We exploit the well-known connection between streaming algorithms and communication complexity to prove linear-space lower bounds for streaming algorithms for LCS and CPM.

5.1 Lower Bounds for Streaming Algorithms

Our streaming lower bounds are based on a reduction from the following problem:

AUGMENTED INDEX
Alice holds a binary string S of length n.
Bob holds an index $i \in [1\,..\,n]$ and the string $S[\,..\,i-1]$.
Output: Bob is to return the value of $S[i]$.

In the one-way communication complexity model, Alice performs an arbitrary computation on her input to create a message $\mathcal{M}$ and sends it to Bob who must compute the output using this message and his input. The communication complexity of a protocol is the size of $\mathcal{M}$ in bits. The protocol is randomised when either Alice or Bob use randomised computation.

▶ **Theorem 5.1** ([18, Theorem 2.3]). *The randomised one-way communication complexity of* AUGMENTED INDEX *is $\Omega(n)$ bits.*

▶ **Theorem 5.2.** *In the streaming setting, any algorithm for LCS for strings of length at most n uses $\Omega(n)$ bits of space.*

Proof. We show the bound by a reduction from the AUGMENTED INDEX problem. Consider an input $S, (i, S[\,..\,i-1])$ to the AUGMENTED INDEX problem, where $|S| = n$. We observe that for $A = 0^n\$S$ and $B = 0^n\$S[\,..\,i-1]1$, where $\$ \notin \{0,1\}$, we have $\mathsf{LCS}(A,B) = n+i+1$ if and only if $S[i] = 1$. Now, if we have a streaming algorithm for LCS that uses b bits of space, we can develop a one-way protocol for the AUGMENTED INDEX problem with message size b bits as follows. Alice runs the algorithm on A. When she reaches the end of A, she sends the memory state of the algorithm and n (in binary) to Bob. Bob continues running the algorithm on B, which he can construct knowing n and $S[\,..\,i-1]$, and returns 1 if and only if $\mathsf{LCS}(A,B) = n+i+1$. Theorem 5.1 implies that $b + \log n = \Omega(n)$, and hence $b = \Omega(n)$. ◀

▶ **Theorem 5.3.** *In the streaming setting, any algorithm for* CPM *uses $\Omega(m)$ bits of space, where m is the size of the pattern.*

Proof. We show the bound by a reduction from the Augmented Index problem. Consider an input $S, (i, S[..i-1])$ to the Augmented Index problem, where $|S| = m$. Let $A = S\$$ and $B = S\$S[..i-1]1$, where $\$ \notin \{0,1\}$. B ends with an occurrence of a rotation of A if and only if $S[i] = 1$. Now, if we have a streaming algorithm for CPM that uses b bits of space, we can develop a one-way protocol for the Augmented Index problem with message size b bits as follows. Alice runs the algorithm on the pattern $A = S\$$ and the first $|S|+1$ letters of the string B. She then sends the memory state of the algorithm to Bob. Bob continues running the algorithm on the remainder of B, i.e., on $S[1..i-1]1$, and returns 1 if and only if the algorithm reports an occurrence of a rotation of A ending at position $n+i+1$. By Theorem 5.1, we have $b = \Omega(m)$. ◀

6 LCS and CPM in the Asymmetric Streaming Setting

In this section, we use Theorem 3.7 to show that for any $\tau \in [\tilde{\Omega}(\sqrt{m})\,..\,O(m/\log^2 m)]$, there are asymmetric streaming algorithms for LCS and CPM that use $O(\tau)$ space and $\tilde{O}(m/\tau)$ time per letter. We start by giving an algorithm for a generalization of the LCS problem that can be used to solve both LCS and CPM. For two strings S, T, a fragment $T[t..t']$ is a *T-maximal common substring* of S and T if it is a occurs in S and neither $T[t-1..t']$ (assuming $t > 1$) nor $T[t..t'+1]$ (assuming $t' < n$) occurs in S.

▶ **Theorem 6.1.** *Assume to be given read-only random access to a string S of length m and streaming access to a string T of length n over an integer alphabet, where $n \geq m$. For all $\tau \in [\sqrt{m}\log m(\log\log m)^3\,..\,O(m/\log^2 m)]$, there is an algorithm that reports all T-maximal common substrings of S and T using $O(\tau)$ space and $O(nm/\tau \cdot \log\log\sigma)$ time.*

Proof. We cover T with windows of length 2τ (except maybe for the last) that overlap by τ letters: there are $O(n/\tau)$ such windows. After reading such a window W, we apply the procedure encapsulated in the following claim with $A = W$ and $B = S$:

▷ Claim 6.2. Let A, B be strings of respective lengths a and b, where $a < b < a^{O(1)}$, over an integer alphabet of size σ. Given read-only random access to A and B, we can compute all B-maximal common substrings of A and B, and the length $\mathsf{LCSuf}(A, B)$ of the longest suffix of A that occurs in B in $O(b\log\log\sigma)$ time using $O(a)$ extra space.

Proof. We start by building the suffix tree for A and preprocessing it for constant-time weighted ancestor queries: this takes $O(a)$ time (see Fact 2.2 and Fact 2.8). Additionally, we preprocess the labels of edges outgoing from each node according to Fact 2.6. Then, the algorithm traverses the tree maintaining the following invariant: at every moment, it is at a node (maybe implicit) corresponding to a substring $B[i..j]$ of B. It starts at the root of the tree with $i = 1$ and $j = 0$. In each iteration, the algorithm tries to go down the tree from the current node using $B[j+1]$; this takes $O(\log\log\sigma)$ time. If it succeeds, it increments j and continues. Otherwise, it considers two cases. If it is at the root, it increments both i and j. Otherwise, it jumps to the node corresponding to $B[i+1..j]$ via a weighted ancestor query in $O(1)$ time and increments i. The nodes reached by an edge traversal and abandoned with the use of a weighted ancestor query in the next iteration are in one-to-one correspondence with the B-maximal common substrings of A and B. The LCSuf of A and B is the depth of the deepest visited node that corresponds to a suffix of A. As at least one of the indices i, j gets incremented at every step of the traversal, the total runtime is $O(b\log\log\sigma)$. ◁

The above sliding-window procedure takes $O(m \log\log\sigma)$ time per window and uses $O(\tau)$ space, which adds up to $O(nm/\tau \cdot \log\log\sigma)$ time in total, and finds all T-maximal common substrings of S and T that have length at most τ.

We run another procedure in parallel in order to compute T-maximal common substrings of length at least τ. During preprocessing, we build the Extended IPM (Decision) data structure (Theorem 3.7) for the string S with $\ell = \tau - 2$ in $O(m \log_{m/\tau} m) = O(nm/\tau)$ time using $O((m/\tau) \cdot \log m (\log\log m)^3) = O(\tau)$ space.

Assume that while reading a window $W = T[\ell\,..\,r]$, the sliding-window procedure found an LCSuf $T[i\,..\,r]$ of length at least τ. We start a search for a common substring starting in W. Let $j \geq r$ be the current letter of T, and $T[i\,..\,j]$, $\ell \leq i \leq r$, be the longest suffix of $T[\ell\,..\,j]$ that occurs in S. We assume that we know a position where $T[i\,..\,j]$ occurs in S, which is the case for $j = r$. When $T[j+1]$ arrives, we update i using the following observation:

▶ **Observation 6.3.** *If $T[i\,..\,j]$ is the longest suffix of $T[1\,..\,j]$ that occurs in S, and $T[i'\,..\,j+1]$ is the longest suffix of $T[1\,..\,j+1]$ that occurs in S, then $i \leq i'$.*

By using binary search and IPM queries, we can find the smallest $i \leq i'$ such that $T[i'\,..\,j+1]$ occurs in S and a witness occurrence, if the corresponding string has length at least τ: namely, if $S[x\,..\,x']$ is a witness occurrence of $T[i\,..\,j]$ in S, we search for occurrences of $P = S[x + (i'-i)\,..\,x']T[j+1]$ in S. If $j - i' < \tau$, we stop the search, and otherwise we set $i' = i$ and continue. It is evident that all T-maximal common substrings of S and T that are of length greater than τ can be extracted during the execution of the above procedure: a maintained suffix of length greater than τ is such a fragment if the last update to it was an increment of its right endpoint, while the next update is an increment of its left endpoint. For every letter, we run at most one binary search which uses $O(\log m)$ IPM queries and hence takes $O(\log m (\log\log m)^3)$ time. As $\tau = O(m/\log^2 m)$, the m/τ term dominates the per-letter running time. The correctness of the described procedure follows from the fact that any substring of T of length greater than τ is either fully contained in the first window or crosses the boundary of some window. ◀

▶ **Corollary 6.4.** *Assume to be given random access to an m-length string S and streaming access to a n-length string T, where $n \geq m$. For all $\tau \in [\sqrt{m} \log m (\log\log m)^3\,..\,O(m/\log^2 m)]$, there is an algorithm that computes LCS(S,T) using $O(nm/\tau \cdot \log\log\sigma)$ time and $O(\tau)$ space.*

Proof. Note that the longest common substring of S and T is a T-maximal substring of S and T. We use the algorithm of Theorem 6.1 with the same value of τ to iterate over all T-maximal common substrings $T[t\,..\,t']$ of S and T, and store the pair of indices t, t' that maximizes $t' - t$. ◀

▶ **Corollary 6.5.** *Assume to be given random access to an m-length pattern P and streaming access to an n-length text T, where $n \geq m$. For all $\tau \in [\sqrt{m} \log m (\log\log m)^3\,..\,O(m/\log^2 m)]$, there is an algorithm that solves the CPM problem for P, T using $O(m/\tau \cdot \log\log\sigma)$ time per letter of T and $O(\tau)$ space.*

Proof. We use the algorithm of Theorem 6.1 with threshold τ on the string $P \cdot P$, to which we have random access, and a streaming string T. The occurrence of any rotation of P in T implies a common substring of $P \cdot P$ and T of length $m \geq 2\tau$. The algorithm of Theorem 6.1 allows us to find such occurrences in $O(m/\tau \cdot \log\log\sigma)$ amortized time per letter of T using $O(\tau)$ space. By noticing that none of the m-length substrings are fully contained in $T(|T| - \tau\,..\,|T|]$, we can deamortise the algorithm using the standard time-slicing technique, cf [30]. ◀

7 CPM in the Read-only Setting

In this section, we present a deterministic read-only online algorithm for the CPM problem.

▶ **Theorem 7.1.** *There is a deterministic read-only online algorithm that solves the CPM problem on a pattern P of length m and a text T of length n using $O(1)$ space and $O(1)$ time per letter of the text.*

Proof. In this proof, we assume that $n \le 2m-1$. If this is not the case, we can cover T with $2m$-length windows overlapping by $m-1$ letters, and process the text window by window; the last window might be shorter. Every occurrence of a rotation of P belongs to exactly one of the windows and hence will be reported exactly once.

We partition P into four fragments P_1, P_2, P_3, P_4, each of length either $\lfloor m/4 \rfloor$ or $\lceil m/4 \rceil$.[2] By applying Fact 2.4, we compute the periods of each of P and P_i for $i \in [1\mathinner{.\,.} 4]$, if it is are periodic. We also compute, for each $i \in [1\mathinner{.\,.} 4]$, the occurrences of P_i in P^2 using Fact 2.3, and store them in $O(1)$ space due to Fact 2.1. Overall, the preprocessing step takes $O(m)$ time and uses constant space.

We compute all occurrences of all P_i in T in an online manner using Fact 2.3. Due to Fact 2.1, we can represent all computed occurrences of each P_i using a constant number of arithmetic progressions with difference $\mathsf{per}(P_i)$ in $O(1)$ space.

▶ **Observation 7.2.** *Assume that $T(j-m\mathinner{.\,.} j] = P[\Delta+1\mathinner{.\,.} m] \circ P[\mathinner{.\,.}\Delta]$. There is an occurrence of P_i at a position ℓ of T such that $j-m < \ell \le j-|P_i|+1$ if and only if there is an occurrence of P_i at position $p = \Delta+\ell-j+m$ of P^2.*

Now, note that for every rotation P' of P, some P_i occurs at one of the first $\phi := 2\lceil m/4 \rceil$ positions of P'. We will use such occurrences as anchors to compute the occurrences of rotations of P in T. Fix i such that there is an occurrence of P_i in the first ϕ positions of $T(j-m\mathinner{.\,.} j]$. We consider two cases depending on whether the period of P_i is large or small.

Case I: $\mathsf{per}(P_i) > |P_i|/4$. By Fact 2.1, there are $O(1)$ occurrences of P_i in each of T and P^2. Suppose that P_i occurs at position ℓ of T. If $T(j-m\mathinner{.\,.} j] = P[\Delta+1\mathinner{.\,.} m] \circ P[\mathinner{.\,.}\Delta]$ for some Δ, then, by Observation 7.2, P_i occurs at position $p = \Delta+\ell-j+m$ of P^2 and we must have that the length of the longest common suffix of $T[1\mathinner{.\,.}\ell)$ and $P^2[1\mathinner{.\,.} p)$ is at least $\ell-(j-m)$ and the length of the longest common prefix of $T[\ell+|P_i|\mathinner{.\,.}]$ and $P^2[p+|P_i|\mathinner{.\,.}]$ is at least $j-\ell-|P_i|$. As we only need to consider occurrences of P_i in the first ϕ positions of rotations of P, we can work under the assumption that $\ell-(j-m) \le \phi$. Hence, it suffices to compute, for every occurrence of P_i at a position p in P^2 and every occurrence of P_i at a position ℓ in T, values

- $x := \max\{\phi, \mathsf{LCE}^R(T[1\mathinner{.\,.}\ell), P^2[1\mathinner{.\,.} p))\}$, the maximum of ϕ and the length of the longest common suffix of $T[1\mathinner{.\,.}\ell)$ and $P^2[1\mathinner{.\,.} p)$;
- $y := \mathsf{LCE}^R(T[1\mathinner{.\,.}\ell), P^2[1\mathinner{.\,.} p))$, the length of the longest common prefix of $T[\ell+|P_i|\mathinner{.\,.}]$ and $P^2[p+|P_i|\mathinner{.\,.}]$.

The length y is computed naively as new letters arrive, while, in order to compute x, we perform a constant number of letter comparisons for each letter that arrives. Since $\ell-(j-m) = O(j-\ell-|P_i|)$, we will have completed the extension to the left when the j-th letter of the text arrives. As there is a constant number of pairs (p,ℓ) to be considered, we perform a total number of $O(1)$ letter comparisons per letter of the text.

[2] The sole reason for partitioning P into four fragments instead of two is to guarantee that there is an occurrence of some P_i close the the starting position of each rotation of P. This allows us to obtain a worst-case rather than an amortised time bound for processing each letter of the text.

Case II: $\mathsf{per}(P_i) \le |P_i|/4$. For brevity, denote $\rho = \mathsf{per}(P_i)$. Below, when we talk about arithmetic progressions of occurrences of P_i, we mean maximal arithmetic progressions of starting positions of occurrences of P_i with difference ρ. Consider the first element ℓ and the last element r of the rightmost computed arithmetic progression of occurrences of P_i in $T(j-m\,..\,j]$. We next distinguish between two cases depending on whether $\mathsf{per}(T(j-m\,..\,j]) = \rho$. This information can be easily maintained in $O(1)$ time per letter using $O(1)$ space as follows. In particular, for each arithmetic progression of occurrences of P_i in T, we perform at most $\rho - 1$ letter comparisons to extend the periodicity to the left; we can do this lazily upon computing the first element of each progression, by performing at most one letter comparison for each of the next $\rho - 1$ letter arrivals. Further, as at most one arithmetic progression corresponds to occurrences of P_i in T that contain a position in $(j-\rho\,..\,j]$, the extensions to the right take $O(1)$ time per letter as well.

Subcase (a): $\mathbf{per}(T(j-m\,..\,j]) \neq \rho$**.** Suppose that $T(j-m\,..\,j] = P[\Delta+1\,..\,m] \circ P[..\,\Delta]$ for some Δ. Then, due to Observation 7.2, one of the two following holds:

1. ℓ and $p_\ell = \Delta + \ell - j + m$ are the first elements in arithmetic progressions of occurrences of P_i in $T(j-m\,..\,j]$ and P^2, respectively;
2. r and $p_r = \Delta + r - j + m$ are the last elements in arithmetic progressions of occurrences of P_i in $T(j-m\,..\,j]$ and P^2, respectively.

We handle this case by considering a subset of pairs of occurrences of P_i and treating them similarly to Case I. Namely, we consider (a) pairs that are first in their respective arithmetic progressions in P^2 and T and (b) pairs that are last in their respective arithmetic progressions in P^2 and $T(j-m\,..\,j]$. By Fact 2.1, there are only a constant number of such elements in P^2 and a constant number of such elements in the text at any time (a previously last element in the text may stop being last when a new occurrence of P_i is detected). For pairs of first elements there are no changes required to the algorithm for Case I. We next argue that, for each pair (r, p_r) of last elements, it suffices to perform only $O(\rho)$ letter comparisons to check how far the periodicity extends to the left, and that this is all we need to check. Due to this, we do not restrict our attention to the case when $r \in (j-m\,..\,j-m+\phi]$, but rather consider all last elements of arithmetic progressions. Let ℓ' be the first element of the arithmetic progression in $T(j-m\,..\,m]$ that contains r. If $\ell' > \rho + j - m$, we avoid extending to the left since either $\ell' \in (j-m\,..\,j-m+\phi]$ and the sought occurrence of a rotation of P, if it exists, will be computed by the algorithm when it processes pair $(\ell', \Delta + \ell' - j + m)$ or the sought occurrence will be computed when processing a different arithmetic progression of occurrences of P_i or a different P_j. Further note that the extension to the left has been already computed; either ℓ' is not the first element in the arithmetic progression of occurrences of P_i in T (we have assumed that it is in $T(j-m\,..\,j]$), in which case we are trivially done, or ℓ' is the first element of an arithmetic progression in T and hence we extended the periodicity via a lazy computation when the occurrence of P_i at position ℓ' was detected. As the occurrences of P_i in T are spaced at least ρ positions away, the above procedure takes $O(1)$ time per letter of the text.

Subcase (b): $\mathbf{per}(T(j-m\,..\,j]) = \rho$**.** Using $O(m)$ time and $O(1)$ extra space, we can precompute all $1 \le j \le \rho$ such that $Q_i^\infty[j\,..\,j+m)$ occurs in P^2, where $Q_i = P_i[1\,..\,\rho]$; it suffices to extend the periodicity for each of the $O(1)$ arithmetic progressions of occurrences of P_i in P^2 and to perform standard arithmetic. In particular, the output consists of a constant number of intervals. Then, if $\mathsf{per}(T(j-m\,..\,j]) = \rho$, $T(j-m\,..\,j]$ equals a rotation of P if and only if $\ell - (j-m) \pmod{\rho}$ is in one of the computed intervals and this can be checked in constant time. ◀

References

1 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11), 2020. doi:10.3390/a13110276.

2 Amihood Amir, Mika Amit, Gad M. Landau, and Dina Sokol. Period recovery of strings over the Hamming and edit distances. *Theoretical Computer Science*, 710:2–18, 2018. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). doi:https://doi.org/10.1016/j.tcs.2017.10.026.

3 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *Proc. of ESA*, pages 5:1–5:18, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

4 Amihood Amir, Ayelet Butman, Eitan Kondratovsky, Avivit Levy, and Dina Sokol. Multidimensional period recovery. *Algorithmica*, 84(6):1490–1510, 2022. doi:10.1007/S00453-022-00926-Y.

5 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

6 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algor.*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

7 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proc. of FOCS*, pages 377–386, 2010. doi:10.1109/FOCS.2010.43.

8 Lorraine A.K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. doi:https://doi.org/10.1016/j.patrec.2017.01.018.

9 Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proc. of SODA*, pages 572–591, 2015. doi:10.1137/1.9781611973730.39.

10 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theoretical Computer Science*, 922:271–282, 2022. doi:https://doi.org/10.1016/j.tcs.2022.04.029.

11 Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern matching with mismatches and wildcards. *CoRR*, abs/2402.07732, 2024. doi:10.48550/ARXIV.2402.07732.

12 Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya. Small-space algorithms for the online language distance problem for palindromes and squares. In *Proc. of ISAAC*, pages 10:1–10:17, 2023. doi:10.4230/LIPICS.ISAAC.2023.10.

13 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *Proc. of CPM*, pages 8:1–8:15, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

14 Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In *Proc. of CPM*, pages 5:1–5:14, 2020. doi:10.4230/LIPICS.CPM.2020.5.

15 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space tradeoffs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/J.JDA.2013.06.003.

16 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proc. of SODA*, pages 607–626, 2020. doi:10.1137/1.9781611975994.37.

17 Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theoretical Computer Science*, 483:2–9, 2013. doi:10.1016/J.TCS.2012.11.040.

18 Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM J. Comput.*, 42(1):61–83, 2013. doi:10.1137/100816481.

19 Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the RAM, revisited. In *Proc. of SoCG*, pages 1–10, 2011. doi:10.1145/1998196.1998198.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *Proc. of CPM*, pages 8:1–8:15, 2020. doi:10.4230/LIPICS.CPM.2020.8.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

22 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *Proc. of ESA*, pages 30:1–30:17, 2021. Full version: https://arxiv.org/abs/2105.03106. doi:10.4230/LIPICS.ESA.2021.30.

23 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular pattern matching with k mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.

24 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In *Proc. of ESA*, pages 35:1–35:19, 2022. doi:10.4230/LIPICS.ESA.2022.35.

25 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Efficient enumeration of distinct factors using package representations. In *Proc. of SPIRE*, volume 12303, pages 247–261. Springer, 2020. doi:10.1007/978-3-030-59212-7_18.

26 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. of FOCS*, pages 978–989, 2020. doi:10.1109/FOCS46700.2020.00095.

27 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *Proc. of FOCS*, pages 698–707, 2022. doi:10.1109/FOCS54457.2022.00072.

28 Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching under edit distance. In *Proc. of STACS*, pages 24:1–24:22, 2024. doi:10.4230/LIPIcs.STACS.2024.24.

29 Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *The Computer Journal*, 57(5):731–743, 03 2013. doi:10.1093/comjnl/bxt023.

30 Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. *Inf. Comput.*, 209(4):731–736, 2011. doi:10.1016/J.IC.2010.12.007.

31 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Internal quasiperiod queries. In *Proc. of SPIRE*, pages 60–75, 2020. doi:10.1007/978-3-030-59212-7_5.

32 Jiangqi Dai, Qingyu Shi, and Tingqiang Xu. Faster algorithms for internal dictionary queries. *CoRR*, abs/2312.11873, 2023. doi:10.48550/ARXIV.2312.11873.

33 Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for Dyck edit distance and RNA folding. In *Proc. of ICALP*, pages 49:1–49:20, 2022. doi:10.4230/LIPIcs.ICALP.2022.49.

34 Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, and Kaiyu Wu. Internal masked prefix sums and its connection to fully internal measurement queries. In *Proc. of SPIRE*, pages 217–232, 2022. doi:10.1007/978-3-031-20643-6_16.

35 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

36 Henning Fernau, Florin Manea, Robert Mercaş, and Markus L. Schmid. Pattern matching with variables: Efficient algorithms and complexity results. *ACM Trans. Comput. Theory*, 12(1), feb 2020. doi:10.1145/3369935.

37 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.

38 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. of ESA*, volume 9294, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.

39 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. of CPM*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

40 Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009. doi:https://doi.org/10.1016/j.jda.2008.09.001.

41 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin 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(1):162–191, 2018. doi:10.1007/S00224-017-9794-5.

42 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, and Sharma V. Thankachan. Space-efficient construction algorithm for the circular suffix tree. In *Proc. of CPM*, pages 142–152, 2013. doi:10.1007/978-3-642-38905-4_15.

43 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Proc. of ISAAC*, pages 673–682, 2011. doi:10.1007/978-3-642-25591-5_69.

44 Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Linear-time computation of cyclic roots and cyclic covers of a string. In *Proc. of CPM*, pages 15:1–15:15, 2023. doi:10.4230/LIPICS.CPM.2023.15.

45 Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and indexing circular patterns. In *Algorithms for Next-Generation Sequencing Data: Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.

46 Marek Karpinski and Yakov Nekrich. Space efficient multi-dimensional range reporting. In *Proc. of COCOON*, volume 5609, pages 215–224. Springer, 2009. doi:10.1007/978-3-642-02882-3_22.

47 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

48 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *Proc. of FOCS*, pages 1002–1013, 2020. doi:10.1109/FOCS46700.2020.00097.

49 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. of STOC*, pages 1657–1670, 2022. Full version at http://arxiv.org/abs/1910.10631. doi:10.1145/3519935.3520061.

50 Tomasz Kociumaka. *Efficient data structures for internal queries in texts.* PhD thesis, University of Warsaw, Warsaw, Poland, October 2018. Available at https://depotuw.ceon.pl/handle/item/3614.

51 Tomasz Kociumaka, Ritu Kundu, Manal Mohamed, and Solon P. Pissis. Longest unbordered factor in quasilinear time. In *Proc. of ISAAC*, pages 70:1–70:13, 2018. doi:10.4230/LIPIcs.ISAAC.2018.70.

52 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Optimal data structure for internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235, 2013. arXiv:1311.6235.

53 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proc. of SODA*, pages 532–551, 2015. doi:10.1137/1.9781611973730.36.

54 Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Proc. of ESA*, pages 605–617, 2014. doi:10.1007/978-3-662-44777-2_50.

55 Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. *Journal of Discrete Algorithms*, 46-47:1–15, 2017. doi:https://doi.org/10.1016/j.jda.2017.10.004.

56 Dmitry Kosolobov, Florin Manea, and Dirk Nowotka. Detecting one-variable patterns. In *Proc. of SPIRE*, pages 254–270, 2017. doi:10.1007/978-3-319-67428-5_22.

57 Dmitry Kosolobov and Nikita Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. In *Proc. of CPM*, 2024.

58 Moshe Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302, 2013. doi:10.1007/978-3-642-40273-9_18.

59 M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005.

60 Tung Mai, Anup Rao, Ryan A Rossi, and Saeed Seddighin. Optimal space and time for streaming pattern matching. *arXiv preprint arXiv:2107.04660*, 2021.

61 Kazuki Mitani, Takuya Mieno, Kazuhisa Seto, and Takashi Horiyama. Internal longest palindrome queries in optimal time. In *Proc. of WALCOM*, pages 127–138, 2023.

62 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. of ICALP*, volume 5125, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

63 Michael Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *Proc. of SODA*, pages 1698–1709, 2013. doi:10.1137/1.9781611973105.122.

64 Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proc. of CPM*, pages 223–234, 2013. doi:10.1007/978-3-642-38905-4_22.

65 Robert Susik, Szymon Grabowski, and Sebastian Deorowicz. Fast and simple circular pattern matching. In *Man-Machine Interactions 3*, pages 537–544, 2014.

66 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.