



Small-Space Algorithms for the Online Language Distance Problem for Palindromes and Squares

Gabriel Bathie  

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

Tomasz Kociumaka  

Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

Tatiana Starikovskaya  

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

Abstract

We study the online variant of the language distance problem for two classical formal languages, the language of palindromes and the language of squares, and for the two most fundamental distances, the Hamming distance and the edit (Levenshtein) distance. In this problem, defined for a fixed formal language L , we are given a string T of length n , and the task is to compute the minimal distance to L from *every* prefix of T . We focus on the low-distance regime, where one must compute only the distances smaller than a given threshold k . In this work, our contribution is twofold:

1. First, we show *streaming* algorithms, which access the input string T only through a single left-to-right scan. Both for palindromes and squares, our algorithms use $O(k \text{ polylog } n)$ space and time per character in the Hamming-distance case and $O(k^2 \text{ polylog } n)$ space and time per character in the edit-distance case. These algorithms are randomised by necessity, and they err with probability inverse-polynomial in n .
2. Second, we show *deterministic read-only* online algorithms, which are also provided with read-only random access to the already processed characters of T . Both for palindromes and squares, our algorithms use $O(k \text{ polylog } n)$ space and time per character in the Hamming-distance case and $O(k^4 \text{ polylog } n)$ space and amortised time per character in the edit-distance case.

2012 ACM Subject Classification Theory of computation \rightarrow Streaming, sublinear and near linear time algorithms; Theory of computation \rightarrow Pattern matching

Keywords and phrases Approximate pattern matching, streaming algorithms, palindromes, squares

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2023.10

Related Version *Full Version with Details for the Edit Distance Algorithms:*
<https://arxiv.org/abs/2309.14788>

Funding Partially funded by the grant ANR-20-CE48-0001 from the French National Research Agency.

1 Introduction

The *language distance* problem is one of the most fundamental problems in formal language theory. In this problem, the task is to compute the minimal distance between a given string S and any string belonging to a formal language L . Introduced in the early 1970s by Aho and Peterson [2], the language distance problem has been studied extensively for regular languages under Hamming and edit distances [5], for general context-free languages, mainly focusing on the edit distance [1, 2, 8, 10, 25, 27, 29, 32, 33, 34], and the Dyck language (the language of well-nested parentheses sequences) in particular [1, 4, 8, 10, 12, 13, 15, 22, 23, 31, 32, 33].



© Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya;
licensed under Creative Commons License CC-BY 4.0

34th International Symposium on Algorithms and Computation (ISAAC 2023).

Editors: Satoru Iwata and Naonori Kakimura; Article No. 10; pp. 10:1–10:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Online Language Distance Problem for Palindromes and Squares

Our results. In this work, we study the complexity of the online and low-distance version of the language distance problem, where we are given a string T of length n , and the task is to compute the minimal distance from *every* prefix of T to a formal language L (the distance and the language are specified in the problem definition). We focus on the low-distance regime, i.e., we assume to be given a threshold parameter k such that distances larger than k do not need to be computed. We consider the edit distance (defined as the minimum number of character insertions, deletions, and substitutions needed to transform one string into the other) and, as a preliminary step, the Hamming distance (allowing for substitutions only). We study the problem for two classical languages: the language PAL of all palindromes, where a palindrome is a string that is equal to its reversed copy, and the language SQ of all squares, where a square is the concatenation of two copies of a string. These two languages are very similar yet very different in nature: PAL is not regular but is context-free, whereas SQ is not even context-free. Formally, the problems we consider are defined as follows:

► **Problem 1.1.** k -LHD-PAL (resp. k -LHD-SQ)

Input: A string T of length n and a positive integer k .

Output: For each $1 \leq i \leq n$, report $\min\{k + 1, hd_i\}$, where hd_i is the minimum Hamming distance between $T[1..i]$ and a string in PAL (resp. in SQ).

► **Problem 1.2.** k -LED-PAL (resp. k -LED-SQ)

Input: A string T of length n and a positive integer k .

Output: For each $1 \leq i \leq n$, report $\min\{k + 1, ed_i\}$, where ed_i is the minimum edit distance between $T[1..i]$ and a string in PAL (resp. in SQ).

■ **Table 1** Summary of the complexities of the algorithms introduced in this work.

Problem	Model	Time per character	Space complexity	Reference
k -LHD-PAL	Streaming	$O(k \log^3 n)$	$O(k \log n)$	Thm 3.2
k -LHD-SQ	Streaming	$\tilde{O}(k)$	$O(k \log^2 n)$	Thm 3.3
k -LHD-PAL/SQ	Read-only	$O(k \log n)$	$O(k \log n)$	Thms 4.8 and 4.10
k -LED-PAL/SQ	Streaming	$\tilde{O}(k^2)$	$\tilde{O}(k^2)$	Thms 5.1 and 5.2
k -LED-PAL/SQ	Read-only	$\tilde{O}(k^4)$ (amortised)	$\tilde{O}(k^4)$	Thms 5.3 and 5.4

Amir and Porat [3] showed that there is a randomised streaming algorithm that solves the k -LHD-PAL problem in $\tilde{O}(k)$ space and $\tilde{O}(k^2)$ time per input character.¹ We continue their line of research and show streaming algorithms for all four problems that use $\text{poly}(k, \log n)$ time per character and $\text{poly}(k, \log n)$ space. While streaming algorithms are extremely efficient (in particular, the space complexities above account for *all* the space used by the algorithms, including the space needed to store information about the input), it is important to note that they are randomised in nature, which means that they may produce incorrect results with a certain probability (inverse polynomial in the input size n). Motivated by this, we also study the problems in the read-only model, where random access to the input is allowed (and not accounted for in the space usage). In this model, we show *deterministic* algorithms for the four problems that use $\text{poly}(k, \log n)$ time per character and $\text{poly}(k, \log n)$ space; see Table 1 for a summary. As a side result of independent interest, we develop the first $\text{poly}(k, \log n)$ space read-only algorithms for computing k -mismatch and k -edit occurrences of a pattern in a text.

¹ Hereafter, $\tilde{O}(\cdot)$ hides factors polynomial in $\log n$.

Due to the lack of space, descriptions of the algorithms for the Language edit distance problems (k -LED-PAL and k -LED-SQ) are omitted from this version of the paper, but can be found in the full one.

1.1 Related work

Offline model. In the classical *offline* model, the problem of finding *all maximal substrings* that are within Hamming distance k from PAL can be solved in $O(nk)$ time as a simple application of the kangaroo jumps technique [17]. For the edit distance, Porto and Barbosa [28] showed an $O(nk^2)$ solution. For the SQ language, the best known solutions take $O(nk \log k + \text{output})$ time for the Hamming distance [21] and $O(nk \log^2 k + \text{output})$ for the edit distance [24, 35, 36].

Online model. The problems k -LHD-PAL and k -LED-PAL can be viewed as a generalization of the classical online palindrome recognition problem (see [16] and references therein).

Streaming algorithms for PAL and SQ. Berebrink et al. [6] followed by Gawrychowski et al. [18] studied the question of computing the length of a maximal substring of a stream that belongs to PAL. Merkurev and Shur [26] considered a similar question for the SQ language.

2 Preliminaries

We assume to be given an alphabet Σ , the elements of which, called *characters*, can be stored in a single machine word of the RAM model. For an integer $n \geq 0$, we denote the set of all length- n strings by Σ^n , and we set $\Sigma^{\leq n} = \bigcup_{m=0}^n \Sigma^m$ as well as $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$. The empty string is denoted by ε .

For two strings $S, T \in \Sigma^*$, we use ST or $S \cdot T$ indifferently to denote their concatenation. For an integer $m \geq 0$, the string obtained by concatenating S to itself m times is denoted by S^m ; note that $S^0 = \varepsilon$. A string S is a *square* if there exists a string T such that $S = T^2$.

For a string $T \in \Sigma^n$ and an index $i \in [1..n]$,² the i th character of T is denoted by $T[i]$. We use $|T| = n$ to denote the length of T . For indices $1 \leq i, j \leq n$, $T[i..j]$ denotes the substring $T[i]T[i+1] \cdots T[j]$ of T if $i \leq j$ and the empty string otherwise. When $i = 1$ or $j = n$, we omit these indices, i.e., we write $T[.j] = T[1..j]$ and $T[i.] = T[i..n]$. We extend the above notation with $T[i..j] = T[i..j-1]$ and $T(i..j) = T[i+1..j]$. We say that a string P is a *prefix* of T if there exists $j \in [1..n]$ such that $P = T[.j]$, and a *suffix* of T if there exists $i \in [1..n]$ such that $P = T[i.]$. We use T^R to denote the reverse of T , that is $T^R = T[n]T[n-1] \cdots T[1]$. A string T is a *palindrome* if $T^R = T$.

We define the *forward cyclic rotation* $\text{rot}(T) = T[2] \cdots T[n]T[1]$. In general, a cyclic rotation $\text{rot}^s(T)$ with shift $s \in \mathbb{Z}$ is obtained by iterating rot or the inverse operation rot^{-1} . A non-empty string $T \in \Sigma^n$ is *primitive* if it is distinct from its non-trivial rotations, i.e., if $T = \text{rot}^s(T)$ holds only when n divides s .

Given two strings U, V and two indices $i \in [1..|U|]$ and $j \in [1..|V|]$, the *longest common prefix* (LCP) of $U[i..]$ and $V[j..]$, denoted $\text{LCP}(U[i..], V[j..])$, is the length of the longest string that is a prefix of both $U[i..]$ and $V[j..]$.

² For integers $i, j \in \mathbb{Z}$, denote $[i..j] = \{k \in \mathbb{Z} : i \leq k \leq j\}$, $[i..j) = \{k \in \mathbb{Z} : i \leq k < j\}$, and $(i..j] = \{k \in \mathbb{Z} : i < k \leq j\}$.

Given two non-empty strings U, Q and an operator F defined over pairs of strings, we use the notation $F(U, Q^\infty)$ for the application of F to U and the prefix $Q^\infty = QQ \dots$ that has the same length as U , i.e., $F(U, Q^\infty) = F(U, Q^m[. . |U|])$, where m is any integer such that $|Q^m| \geq |U|$. We define $F(Q^\infty, U)$ symmetrically.

2.1 Hamming distance, palindromes, and squares

The Hamming distance between two strings S, T (denoted $\text{hd}(S, T)$) is defined to be equal to infinity if S and T have different lengths, and otherwise to the number of positions where the two strings differ (mismatches). We define the *mismatch information* between two length- n strings S and T , $\text{MI}(S, T)$ as the set $\{(i, S[i], T[i]) : i \in [1..n] \text{ and } S[i] \neq T[i]\}$. For two strings P, T , a position $i \in [|P|. |T|]$ of T is a k -*mismatch occurrence* of P in T if $\text{hd}(T(i-|P|. i], P) \leq k$. For an integer k , we denote $\text{hd}_{\leq k}(X, Y) = \text{hd}(X, Y)$ if $\text{hd}(X, Y) \leq k$ and ∞ otherwise.

Due to the self-similarity of palindromes and squares, the Hamming distance from a string U to PAL and SQ can be measured in terms of the self-similarity of U .

► **Property 2.1.** *Each string $U \in \Sigma^m$ satisfies $\text{hd}(U, \text{PAL}) = \text{hd}(U[. . \lfloor m/2 \rfloor], U(\lceil m/2 \rceil . .)^R) = \frac{1}{2} \text{hd}(U, U^R)$.*

Proof. Denote $U_1 = U[. . \lfloor m/2 \rfloor]$ and $U_2 = U(\lceil m/2 \rceil . .)^R$. For the second equality, we have $\text{hd}(U, U^R) = \text{hd}(U_1, U_2^R) + \text{hd}(U_2, U_1^R) = 2 \cdot \text{hd}(U_1, U_2^R)$.

The first equality is equivalent to $\text{hd}(U_1, U_2^R) = \text{hd}(U, \text{PAL})$. As the Hamming distance between U and the palindrome $U_2^R U_2$ (or $U_2^R a U_2$ if m is odd) is $\text{hd}(U_1, U_2^R)$, we have $\text{hd}(U_1, U_2^R) \geq \text{hd}(U, \text{PAL})$.

Conversely, let V be a palindrome such that $\text{hd}(U, V) = \text{hd}(U, \text{PAL})$. We decompose similarly V into $V_1 V_1^R$ (or $V_1 b V_1^R$ for odd m) and obtain $\text{hd}(U, V) \geq \text{hd}(U_1, V_1) + \text{hd}(U_2, V_1^R)$. Using the fact that $\text{hd}(U_2, V_1^R) = \text{hd}(U_2^R, V_1)$ and applying the triangle inequality, we get $\text{hd}(U_1, U_2^R) \leq \text{hd}(U, \text{PAL})$. ◀

► **Property 2.2.** *Each string $U \in \Sigma^m$ satisfies $\text{hd}(U, \text{SQ}) = \text{hd}(U[. . m/2], U(m/2 . .))$ if m is even and $\text{hd}(U, \text{SQ}) = \infty$ if m is odd.*

Proof. Every square has even length; hence, if m is odd, the distance between U and SQ is infinite. In what follows, we assume that $m = 2i$ for some $i \in \mathbb{N}$. Let $U_1 = U[. . i]$ and $U_2 = U(i . .)$. By modifying the copy of U_1 in U into U_2 , we obtain a square $U_2 U_2$; hence, $\text{hd}(U, \text{SQ}) \leq \text{hd}(U_1, U_2)$.

For the converse inequality, let V^2 be a square such that $\text{hd}(U, \text{SQ}) = \text{hd}(U, V^2)$. We have $|V| = |U_1| = |U_2|$; hence, $\text{hd}(U, V^2) = \text{hd}(U_1, V) + \text{hd}(V, U_2)$. Applying the triangle inequality, we obtain $\text{hd}(U, \text{SQ}) = \text{hd}(U, V^2) \geq \text{hd}(U_1, U_2)$. ◀

2.2 Models of computation

In this work, we focus on two by now classical models of computation: streaming and read-only random access. In the streaming model, we assume that the input string T arrives as a stream, one character at a time. For each prefix $T[1..i]$, we must report the distance to PAL or SQ as soon as we receive $T[i]$. We account for all the space used, including the space needed to store any information about T . In contrast, in the read-only model, we do not account for the space occupied by the input string. We assume that T is read from the left to the right. After having read $T[1..i]$, we assume to have constant-time read-only random access to the first i characters of T . Similar to the streaming model, the distance from $T[1..i]$ to PAL or SQ must be reported as soon as we read $T[i]$.

3 Warm-up: Streaming algorithms for the LHD problems

In this section, we present streaming algorithms for k -LHD-PAL and k -LHD-SQ. Our solutions use the Hamming distance sketches introduced by Clifford, Kociumaka, and Porat [11] to solve the streaming k -mismatch problem.

► **Fact 3.1.** *There exists a function sk_k^{hd} (parameterized by a constant $c > 1$, integers $n \geq k \geq 1$, and a seed of $O(\log n)$ random bits) that assigns an $O(k \log n)$ -bit sketch to each string in $\Sigma^{\leq n}$. Moreover:*

1. *There is an $O(k \log^2 n)$ -time encoding algorithm that, given $U \in \Sigma^{\leq k}$, builds $\text{sk}_k^{\text{hd}}(U)$.*
2. *There is an $O(k \log n)$ -time algorithm that, given any two among $\text{sk}_k^{\text{hd}}(U)$, $\text{sk}_k^{\text{hd}}(V)$, or $\text{sk}_k^{\text{hd}}(UV)$, computes the third one (provided that $|UV| \leq n$).*
3. *There is an $O(k \log^3 n)$ -time decoding algorithm that, given $\text{sk}_k^{\text{hd}}(U)$ and $\text{sk}_k^{\text{hd}}(V)$, computes $\text{MI}(U, V)$ if $\text{hd}(U, V) \leq k$. The error probability is $O(n^{-c})$.*

3.1 A streaming algorithm for k -LHD-PAL

We first show that the sketches described in Fact 3.1 give a simple algorithm improving upon the result of Amir and Porat [3] and achieving the time complexity of $\tilde{O}(k)$ per letter.

► **Theorem 3.2.** *There is a randomised streaming algorithm that solves the k -LHD-PAL problem for a string $T \in \Sigma^n$ using $O(k \log n)$ bits of space and $O(k \log^3 n)$ time per character. The algorithm errs with probability inverse-polynomial in n .*

Using Property 2.1, we can reduce the k -LHD-PAL problem to that of computing the threshold Hamming distance between the current prefix of the input string and its reverse. The algorithm maintains the sketches $\text{sk}_{2k}^{\text{hd}}(T[. .i])$ and $\text{sk}_{2k}^{\text{hd}}(T[. .i]^R)$. When it receives $T[i]$, it constructs $\text{sk}_{2k}^{\text{hd}}(T[i])$, updates both $\text{sk}_{2k}^{\text{hd}}(T[. .i])$ and $\text{sk}_{2k}^{\text{hd}}(T[. .i]^R)$, and computes $d = \text{hd}_{\leq 2k}(T[. .i], T[. .i]^R)$ (in $O(k \log^3 n)$ total time by Fact 3.1). Property 2.1 implies $\text{hd}_{\leq k}(T[. .i], \text{PAL}) = d/2$. The error probability of the algorithm follows from the error probability for the decoding algorithm for Hamming distance sketches.

The algorithm uses $O(k \log n)$ bits, which is nearly optimal: Indeed, by Property 2.1, if $U = VW$, with $|V| = |W|$, then $\text{hd}(U, U^R) = 2 \cdot \text{hd}(V, W^R)$. Therefore, using a standard reduction from streaming algorithms to one-way communication complexity protocols, we obtain a lower bound of $\Omega(k)$ bits for the space complexity of streaming algorithms for the k -LHD-PAL problem from the $\Omega(k)$ bits lower bound for the communication complexity of the Hamming distance [19].

3.2 A streaming algorithm for k -LHD-SQ

In this section, we show the following theorem:

► **Theorem 3.3.** *There is a randomised streaming algorithm that solves the k -LHD-SQ problem for a string $T \in \Sigma^n$ using $O(k \log^2 n)$ bits of space and $\tilde{O}(k)$ time per character. The algorithm errs with probability inverse-polynomial in n .*

Property 2.2 allows us to derive $\text{hd}_{\leq k}(T[. .2i], \text{SQ})$ from the sketches $\text{sk}_k^{\text{hd}}(T[. .i])$ and $\text{sk}_k^{\text{hd}}(T[. .2i])$: we can combine them to obtain $\text{sk}_k^{\text{hd}}(T(i. .2i))$, and a distance computation on $\text{sk}_k^{\text{hd}}(T[. .i])$ and $\text{sk}_k^{\text{hd}}(T(i. .2i))$ returns $\text{hd}_{\leq k}(T[. .i], T(i. .2i)) = \text{hd}_{\leq k}(T[. .2i], \text{SQ})$.

Naively applying this procedure requires storing the sketch $\text{sk}_k^{\text{hd}}(T[. .i])$ until the algorithm has read $T[. .2i]$, that is, storing $\Theta(n)$ sketches at the same time. To reduce the number of sketches stored, we use a filtering procedure based on the following observation:

► **Observation 3.4.** *If $\text{hd}(T[.2i], SQ) \leq k$ and $\ell \in [1..i]$, then $i + \ell$ is a k -mismatch occurrence of $T[. \ell]$, that is, $\text{hd}(T[. \ell], T(i..i + \ell)) \leq k$.*

► **Example 3.5.** For $k = 1, \ell = 2$, and $i = 3$, the word $T[.6] = \text{abcacc}$ is a 1-mismatch square (by Property 2.2) and the fragment $T(3..5) = \text{ac}$ is a 1-mismatch occurrence of the prefix $T[.2] = \text{ab}$.

Observation 3.4 motivates our filtering procedure: if we choose some prefix $P = T[. \ell]$ of the string, we only need to store every $i \geq \ell$ such that $i + \ell$ is a k -mismatch occurrence of P . Clifford, Kociumaka and Porat [11] showed a data structure \mathcal{S} that exploits the structure of such occurrences and stores them using $O(k \log^2 n)$ bits of space while allowing reporting the occurrence at position $i + \ell$ when $T[i + \ell + \Delta]$ is pushed into \mathcal{S} – we say that \mathcal{S} reports the k -mismatch occurrences of P in T with a *fixed delay* Δ [11]. Our algorithm needs to receive the occurrence at position $i + \ell$ when $T[2i]$ is pushed into the stream, i.e., we require \mathcal{S} to report occurrences with *non-decreasing* delays. In Section 3.2.1 we present a modification of the data structure [11] to allow non-decreasing delays, and in Section 3.2.2 we explain how we use it to implement a space-efficient streaming algorithm for k -LHD-SQ.

3.2.1 Reporting k -mismatch occurrences with nondecreasing delay

The algorithm of Clifford, Kociumaka, and Porat [11] reports additional information along with the positions of the k -mismatch occurrences: specifically, it produces the *stream of k -mismatch occurrences of P in T* , defined as follows.

► **Definition 3.6** ([11, Definition 3.2]). *The stream of k -mismatch occurrences of a pattern P in a text T is a sequence S_P^k such that $S_P^k[i] = (i, \text{MI}(T(i - |P|..i), P), \text{sk}_k^{\text{hd}}(T[.i - |P|]))$ if $\text{hd}(P, T(i - |P|..i)) \leq k$ and $S_P^k[i] = \perp$ otherwise.*

As explained next, the algorithm of [11] can report the k -mismatch occurrences with a prescribed delay.

► **Corollary 3.7** (of [11]). *There is a streaming algorithm that, given a pattern P followed by a text $T \in \Sigma^n$, reports the k -mismatch occurrences of P in T using $O(k \log^2 n)$ bits of space and $O(\sqrt{k} \log^3 n + \log^4 n)$ time per character. The algorithm can report each occurrence i with no delay (that is, upon receiving $T[i]$) or with any prescribed delay $\Delta = \Theta(|P|)$ (that is, upon receiving $T[i + \Delta]$). For each reported occurrence i , the underlying tuple $S_P^k[i]$ can be provided on request in $O(k \log^2 n)$ time.*

Proof. If no delay is required, we use [11, Theorem 1.2], which reports k -mismatch occurrences of P in T and, upon request, provides the mismatch information $\text{MI}(T(i - |P|..i), P)$; this algorithm uses $O(k \log^2 n)$ bits of space and takes $O(\sqrt{k} \log^3 n + \log^4 n)$ time per character. We also use [11, Fact 4.4] to maintain the sketch $\text{sk}_k^{\text{hd}}(T[.i])$ (reported on request); this algorithm uses $O(k \log n)$ bits of space and takes $O(\log^2 n)$ time per character.

Whenever requested to provide $S_P^k[i]$ for some k -mismatch occurrence i of P in T , we retrieve the mismatch information $\text{MI}(T(i - |P|..i), P)$ (in $O(k)$ time) and the sketch $\text{sk}_k^{\text{hd}}(T[.i])$ (in $O(k \log^2 n)$ time). Combining $\text{sk}_k^{\text{hd}}(P)$ with $\text{MI}(T(i - |P|..i), P)$, we build $\text{sk}_k^{\text{hd}}(T(i - |P|..i))$ (using [11, Lemma 6.4] in $O(k \log^2 n)$ time) and then derive $\text{sk}_k^{\text{hd}}(T[.i - |P|])$ using Fact 3.1 (in $O(k \log n)$ time). Overall, processing the request takes $O(k \log^2 n)$ time and $O(k \log^2 n)$ bits of space.

If a delay $\Delta = \Theta(|P|)$ is required, our approach depends on whether there exists $p \in [1..k]$ such that $\text{hd}(P[.|P| - p], P(p..|P|)) \leq 2k$ (such p is called a $2k$ -period in [11]). This property is tested using a streaming algorithm of [11, Lemma 4.3], which takes $O(k \log n)$

bits of space, $O(\sqrt{k \log n})$ time per character of P , and requires $O(k\sqrt{k \log n})$ -time post-processing (performed while reading $T[.k]$). If P satisfies this condition, then we just use [11, Theorem 4.2], whose statement matches that of Corollary 3.7.

Otherwise, [11, Observation 4.1] shows that P has at most one k -mismatch occurrence among any k consecutive positions in T . In that case, we use the aforementioned approach to produce the stream S_P^k with no delay and the buffer of [11, Proposition 3.3] to delay the stream by Δ characters. The buffering algorithm takes $O(k \log^2 n)$ bits of space and processes each character $T[i]$ in $O(k \log^2 n + \log^3 n)$ time (if P has k -mismatch occurrences at positions i or $i - \Delta$) or $O(\sqrt{k \log n} + \log^3 n)$ time (otherwise). Since the former case holds for at most two out of every k consecutive positions, we can achieve $O(\sqrt{k \log^3 n} + \log^4 n)$ worst-case time per character by decreasing the delay to $\Delta - k$ and buffering up to k characters of T and up to k elements of S_P^k . While the algorithm processes $T[i + \Delta]$, the latter buffer already contains $S_P^k[i]$, but $O(k)$ time is still needed to output this value (if $S_P^k[i] \neq \perp$). ◀

The algorithm of Corollary 3.7 has a fixed delay Δ , i.e., it outputs $S_P^k[i]$ upon receiving $T[i + \Delta]$. Our application requires a variable delay: we need to access $S_P^k[i + |P|]$ upon reading $T[2i]$, that is, with a delay of $i - |P|$. We present a black-box construction that extends the data structure of Corollary 3.7 to support non-decreasing delays Δ_i , $i \in [1..d]$. Naively, one could use the algorithm \mathcal{A} of Corollary 3.7 with a fixed delay Δ_1 and buffer the input characters so that \mathcal{A} receives $T[i + \Delta_1]$ only when we actually process $T[i + \Delta_i]$. Unfortunately, this requires storing $T[i + \Delta_1..i + \Delta_i]$, which could take too much space. Thus, we feed \mathcal{A} with $T[1..\Delta_1]$ followed by blank characters \perp (issued at appropriate time steps without the necessity of buffering input characters) so that \mathcal{A} reports k -mismatch occurrences $i \in [1..\Delta_1]$ with prescribed delays. Then, we use another instance of the algorithm of Corollary 3.7, with a fixed delay $\Delta_1 + \Delta_1$, to output k -mismatch occurrences $i \in (\Delta_1..\Delta_1 + \Delta_1 + \Delta_1]$; we continue this way until the whole interval $[1..d]$ is covered. We formalise this idea in the following lemma.

► **Lemma 3.8.** *Let $\Delta_1 \leq \Delta_2 \leq \dots \leq \Delta_d$ be a non-decreasing sequence of $d = O(|P|)$ integers $\Delta_i = \Theta(|P|)$, represented by an oracle that reports each element Δ_i in constant time.*

There is a streaming algorithm that, given a pattern P followed by a text T , reports the k -mismatch occurrences of P in T using $O(k \log^2 n)$ bits of space and $O(\sqrt{k \log^3 n} + \log^4 n)$ time per character. The algorithm reports each occurrence $i \in [1..d]$ with delay Δ_i , that is, upon receiving $T[i + \Delta_i]$. For each reported occurrence $i \in [1..d]$, the underlying tuple $S_P^k[i]$ can be provided on request in $O(k \log n)$ time.

Proof. We use multiple instances $\mathcal{A}_1, \dots, \mathcal{A}_t$ of the algorithm of Corollary 3.7. We define a sequence $(s_r)_{r=0}^t$ so that \mathcal{A}_r works with a fixed delay $\Delta_{s_{r-1}}$, it is given $T[1..s_r] \cdot \perp^{\Delta_{s_{r-1}}}$, and it reports k -mismatch occurrences $i \in [s_{r-1}..s_r)$. Specifically, we set $s_0 = 1$ and $s_r = s_{r-1} + \Delta_{s_{r-1}}$, with t chosen as the smallest integer such that $s_t > d$. Note that $s_r - s_{r-1} = \Delta_{s_{r-1}} \geq \Delta_1$ implies $t \leq 1 + \frac{d}{\Delta_1} = O(1)$.

We assign three different roles to the algorithms $\mathcal{A}_1, \dots, \mathcal{A}_r$: *passive*, *active*, and *inactive*. While we process $T[j]$, the algorithm \mathcal{A}_r is passive if $j < s_r$, active if $j \in [s_r..s_{r+1})$, and inactive if $j \geq s_{r+1}$. Our invariant is that, once we process $T[j]$, each passive algorithm \mathcal{A}_r has already received $T[1..j]$, the unique active algorithm \mathcal{A}_r has already received $T[1..s_r] \cdot \perp^{1+i-s_{r-1}}$, where i is the largest integer such that $i + \Delta_i \leq j$, and each inactive algorithm \mathcal{A}_r has already received its entire input, that is, $T[1..s_r] \cdot \perp^{\Delta_{s_{r-1}}}$.

Upon receiving $T[j]$, we simply forward $T[j]$ to all passive algorithms. Moreover, if $j = i + \Delta_i$ for some $i \in [1..d]$, we feed the active algorithm with \perp so that it checks whether i is a k -mismatch occurrence of P in T and, upon request, outputs $S_P^k[i]$.

Let us argue that this approach is correct from the perspective of a fixed algorithm \mathcal{A}_r . As we process $T[1..s_r)$, the algorithm is passive, and it is fed with subsequent characters of T . For $j = s_r - 1$, the position $i = s_{r-1} - 1$ is the maximum one such that $i + \Delta_i \leq j$. Consequently, the input $T[1..s_r)$ already satisfies the invariant for passive algorithms. For subsequent iterations $j \in [s_r..s_{r+1})$, as \mathcal{A}_r is active, it receives \perp whenever i increases, so its input stays equal to $T[1..s_r) \cdot \perp^{1+i-s_r-1}$. The length of this string is $s_r + i - s_{r-1} = i + \Delta_{s_{r-1}}$, so the algorithm indeed checks whether i is a k -mismatch occurrence of P in T at each such iteration (recall that its fixed delay is $\Delta_{s_{r-1}}$), and it satisfies the invariant for active algorithms. Once we reach $j = s_{r+1} - 1$, we have $i = s_r - 1 = s_{r-1} + \Delta_{s_{r-1}} - 1$, so the input becomes $T[1..s_r) \cdot \perp^{\Delta_{s_{r-1}}}$, and it already satisfies the invariant for inactive algorithms. The state of inactive algorithms does not change, so this invariant remains satisfied as \mathcal{A}_r stays inactive indefinitely.

The time and space complexity analysis follows from the fact that $t = O(1)$. \blacktriangleleft

3.2.2 Algorithm

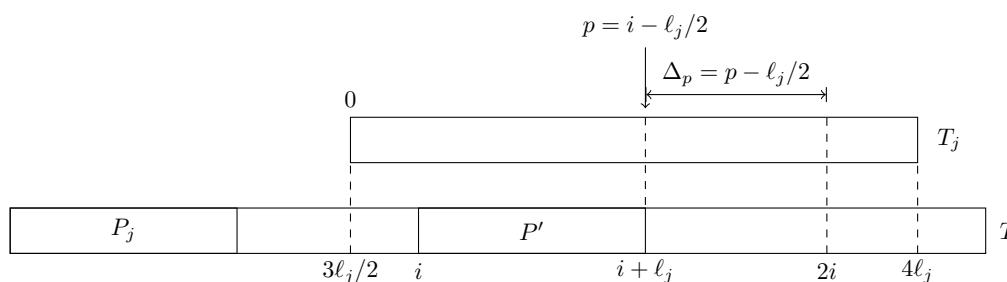
We now show how to use the data structure of Lemma 3.8 to implement our filtering procedure using low space. For each $j \in [1..[\log n]]$, let P_j denote the prefix of the text of length $\ell_j = 2^j$, i.e., $P_j = T[. .2^j]$. We search for k -mismatch occurrences of P_j in $T_j = T(3\ell_j/2..4\ell_j]$. As argued below, this allows filtering positions in $(3\ell_j..6\ell_j]$. Additionally, our choice of $(\ell_j)_j$ ensures that we do not miss any k -mismatch square when running our search for every P_j in parallel.

\triangleright **Claim 3.9.** For each $j \in [1..[\log n]]$, let Occ_j be the set of k -mismatch occurrences of P_j in $T_j = T(3\ell_j/2..4\ell_j]$. If $\text{hd}(T[. .2i], \text{SQ}) \leq k$ and $2i \in [3\ell_j..6\ell_j)$, then $p = i - \ell_j/2 \in \text{Occ}_j$.

Proof. Since $\ell_j \leq i$, Observation 3.4 implies that $i + \ell_j$ is a k -mismatch occurrence of P_j in T . Moreover, when $2i \in [3\ell_j..6\ell_j)$, we have $3\ell_j/2 \leq i \leq 3\ell_j$; therefore, that k -mismatch occurrence of P_j is fully contained within T_j , and it ends at positions $i + \ell_j - 3\ell_j/2 = i - \ell_j/2$ of T_j . \triangleleft

In what follows, we use p to denote indices in T_j , whereas i denotes indices in the original text T . As $T_j = T(3\ell_j/2..4\ell_j]$, the correspondence is given by $i = p + 3\ell_j/2$. In other words, we only need to compute $\text{hd}_{\leq k}(T[. .2i], \text{SQ})$ when $i - \ell_j/2 \in \text{Occ}_j$. As noted in Property 2.2, it suffices to know the sketches $\text{sk}_k^{\text{hd}}(T(i..2i))$ and $\text{sk}_k^{\text{hd}}(T[. .i])$. We store $\text{sk}_k^{\text{hd}}(P_j) = \text{sk}_k^{\text{hd}}(T[. .\ell_j])$ as well as $s_j = \text{sk}_k^{\text{hd}}(T[. .3\ell_j/2])$ and maintain $\text{sk}_k^{\text{hd}}(T[. .2i])$ in a rolling manner as we receive the characters of the text.

We use the algorithm of Lemma 3.8, asking for k -mismatch occurrences of P_j in T_j , to report $\text{sk}_k^{\text{hd}}(T_j[. .i - \ell_j]) = \text{sk}_k^{\text{hd}}(T(\ell_j..i))$ for every $i \in \text{Occ}_j$. The delay sequence is specified as $\Delta_p = p - \ell_j/2$ for $p \in [\ell_j..5\ell_j/2)$ so that the conditions of Lemma 3.8 are satisfied. (For $p < \ell_j$, we can assume $\Delta_p = \Delta_{\ell_j} = \ell_j/2$; anyway, there cannot be a k -mismatch occurrence of P_j before position ℓ_j .) This way, for every $i \in [3\ell_j/2..3\ell_j)$, we receive $S_{P_j}^k[i + \ell_j]$ (which corresponds to a potential k -mismatch occurrence starting at position $i + 1$) while processing $T_j[p + \Delta_p]$ for $p = i + \ell_j - 3\ell_j/2 = i - \ell_j/2$. As $\Delta_p = p - \ell_j/2$, this corresponds to position $p' = 2p - \ell_j/2$ in T_j , or position $i' = 2p + \ell_j = 2i$ in T , i.e., this happens precisely as we are processing $T[2i]$. See Figure 1 for an illustration of the above. If $S_{P_j}^k[i + \ell_j]$ is blank, we move on to the next position. Otherwise, we retrieve the sketch $\text{sk}_k^{\text{hd}}(T_j[. .i]) = \text{sk}_k^{\text{hd}}(T(3\ell_j/2..i))$, combine it with $s_j = \text{sk}_k^{\text{hd}}(T[. .3\ell_j/2])$ and $\text{sk}_k^{\text{hd}}(T[. .2i])$ to obtain $\text{sk}_k^{\text{hd}}(T[. .i])$ and $\text{sk}_k^{\text{hd}}(T(i..2i))$, and use the latter two sketches to compute $\text{hd}_{\leq k}(T[. .i], T(i..2i))$, which is equal to $\text{hd}_{\leq k}(T[. .2i], \text{SQ})$ by Property 2.2.



■ **Figure 1** Illustration of our filtering procedure. Here, P' is a k -mismatch occurrence of P_j at position $i + \ell_j$ in T and position $p = i - \ell_j/2$ in T_j , reported with delay $\Delta_p = p - \ell_j/2$ in T_j , hence it arrives at time $2i$ in T .

We proceed with the complexity analysis of our algorithm. The k -mismatch pattern matching algorithm of Lemma 3.8 uses $O(k \log^2 n)$ bits of space and $\tilde{O}(k)$ time per character, and we maintain $O(\log n)$ instances of this algorithm. However, since all the patterns P_j are prefixes of T , the instances can share the pattern processing phase. Moreover, since any position is contained in at most three fragments $T[\ell_j..6\ell_j]$ (each such fragment follows P_j and contains T_j), at most three instances contribute to the time and space complexity at any given moment. Thus, the entire algorithm uses $O(k \log^2 n)$ bits of space and $\tilde{O}(k)$ time per character, which completes the proof of Theorem 3.3.

Our streaming algorithm for k -LED-SQ (Theorem 5.2) relies on the streaming algorithm for k -LHD-SQ. It requires testing $\text{hd}(T[.2i], \text{SQ}) \leq k$ only for selected positions i , and thus it benefits from the following variant of Theorem 3.3:

► **Proposition 3.10.** *There is a randomised streaming algorithm that, given a string $T \in \Sigma^n$, upon receiving $T[2i]$, can be requested to test whether $\text{hd}(T[.2i], \text{SQ}) \leq k$ and, if so, report the mismatch information between $T[.2i]$ and a closest square. The algorithm uses $O(k \log^2 n)$ bits of space and processes each character in $\tilde{O}(\sqrt{k})$ or $\tilde{O}(k)$ time, depending on whether the request has been issued at that character.*

Proof. We follow the algorithm above with minor modifications. First, instead of maintaining $\text{sk}_k^{\text{hd}}(T[.2i])$ explicitly, we apply [11, Fact 4.4], which uses $O(k \log n)$ bits of space, takes $O(\log^2 n)$ time per character, and reports $\text{sk}_k^{\text{hd}}(T[.2i])$ on demand in $O(k \log^2 n)$ time.

To process a request concerning position $2i$, we retrieve $\text{sk}_k^{\text{hd}}(T[.2i])$ and ask the pattern-matching algorithm of Lemma 3.8 to output $S_{P_j}^k[i]$ (normally, the algorithm only reports whether i is a k -mismatch occurrence of P_j in T_j). In this case, we build $\text{sk}_k^{\text{hd}}(T[.i])$ and $\text{sk}_k^{\text{hd}}(T(i..2i))$ as in algorithm above. The decoding algorithm not only results in $\text{hd}_{\leq k}(T[.i], T(i..2i)) = \text{hd}_{\leq k}(T[.2i], \text{SQ})$ but, if $\text{hd}(T[.2i], \text{SQ}) \leq k$, also the underlying mismatch information.

The space complexity of the modified algorithm is still $O(k \log^2 n)$ bits. The running time is $\tilde{O}(\sqrt{k})$ if we do not ask the algorithm to test $\text{hd}(T[.2i], \text{SQ}) \leq k$ and $\tilde{O}(k)$ if we do. ◀

4 Deterministic read-only algorithms for the LHD problems

In this section, we present deterministic read-only algorithms for k -LHD-PAL and k -LHD-SQ. We start by recalling structural results for k -mismatch occurrences used by the algorithms.

4.1 Structure of k -mismatch occurrences

► **Definition 4.1** ([9]). *A string U is d -mismatch periodic if there exists a primitive string Q such that $|Q| \leq |U|/128d$ and $\text{hd}(U, Q^\infty) \leq 2d$. Such a string Q is called the d -mismatch period of U .*

The condition $|Q| \leq |U|/128d$ implies that Q is equal to some substring of U ; hence, given the starting and ending positions of Q in U and random access to U , we can simulate random access to Q .

▷ **Claim 4.2** (From [20, Claim 7.1]). *Let U and V be strings such that U is a prefix of V , and $|V| \leq 2|U|$. If U is d -mismatch periodic with d -mismatch period Q , then V either is not d -mismatch periodic or has d -mismatch period Q .*

Charalampopoulos, Kociumaka, and Wellnitz [9] showed that the set of k -mismatch occurrences has a very regular structure:

- **Fact 4.3** (See [9, Section 3]). *Let P and T be two strings such that $|P| \leq |T| \leq 3/2|P|$.*
1. *If P is not k -mismatch periodic, then there are $O(k)$ k -mismatch occurrences of P in T .*
 2. *If P is k -mismatch periodic with period Q , then any two k -mismatch occurrences $i \leq i'$ of P in T satisfy $i \equiv i' \pmod{|Q|}$ and $\text{hd}(T(i - |P|.i'), Q^\infty) \leq 3k$.*

They also presented efficient offline algorithms for computing the k -mismatch period and the k -mismatch occurrences in the so-called PILLAR model. In this model, one is given a family of strings \mathcal{X} for preprocessing. The elementary objects are fragments $X[i..j]$ of strings $X \in \mathcal{X}$. Given elementary objects S, S_1, S_2 , the PILLAR operations are:

1. **Access**(S, i): Assuming $i \in [1..|S|]$, retrieve $S[i]$.
2. **Length**(S): Retrieve the length $|S|$ of S .
3. **LCP**(S_1, S_2): Compute the length of the longest common prefix of S_1 and S_2 .
4. **LCP^R**(S_1, S_2): Compute the length of the longest common suffix of S_1 and S_2 .
5. **IPM**(S_1, S_2): Assuming that $|S_2| \leq 2|S_1|$, compute the set of the starting positions of occurrences of S_1 in S_2 , which by Fine and Wilf periodicity lemma [14] can be represented as one arithmetic progression.

In the read-only model, operations **Access** and **Length** can be implemented in constant time and $O(\log m)$ bits. The operations **LCP** and **LCP^R** can be implemented naively via character-by-character comparison in $O(\min\{|S_1|, |S_2|\})$ total time and $O(\log m)$ bits. Finally, the **IPM** operation can be implemented in $O(|S_1| + |S_2|)$ total time and $O(\log m)$ bits (see e.g. [30]).

As a corollary, we immediately obtain:

► **Corollary 4.4** (From [9, Lemma 4.4]). *Given random access to a string U , testing whether it is d -mismatch periodic, and, if so, computing its d -mismatch period, can be done using $O(d|U|)$ time and $O(d)$ space.*

4.2 Read-only algorithm for the pattern matching with k mismatches

The above implementation of the PILLAR operations further implies an offline algorithm that finds all k -mismatch occurrences of P in T in $\tilde{O}(k^2 \cdot |T|)$ time and $\tilde{O}(k^2)$ space (see [9, Main Theorem 8]). Nevertheless, we provide a more efficient online algorithm that additionally provides the mismatch information for every k -mismatch occurrence of P .

► **Theorem 4.5.** *There is a deterministic online algorithm that finds all k -mismatch occurrences of a length- m pattern P within a text T using $O(k \log m)$ space and $O(k \log m)$ worst-case time per character. The algorithm outputs the mismatch information along with every reported k -mismatch occurrence of P .*

Consistently with the streaming algorithm of [11], our algorithm uses a family of exponentially-growing prefixes to filter out candidate positions. However, in order to use the structural properties of Fact 4.3 efficiently, we construct a different family \mathcal{P} to ensure that we are either working in an approximately periodic region of the text or processing an aperiodic prefix.

We first add to \mathcal{P} the prefixes $R_j = P[. \min\{m, \lfloor (3/2)^j \rfloor\}]$ for $j \in [0. \lceil \log_{3/2} m \rceil]$. If R_j is k -mismatch periodic but R_{j+1} is not, we also add to \mathcal{P} the shortest extension of R_j that is not k -mismatch periodic. Hereafter, let $\mathcal{P} = (P_j)_{j=1}^t$ denote the resulting sequence of prefixes, sorted in order of increasing lengths, and let $\ell_j = |P_j|$ for every $j \in [1. t]$.

▷ **Claim 4.6.** The sequence $\mathcal{P} = (P_j)_{j=1}^t$ satisfies the following properties:

- (a) $P_1 = P[1]$ and $P_t = P$,
- (b) $t = |\mathcal{P}| = O(\log m)$,
- (c) for every $j \in [1. t]$, we have $\ell_{j+1} \leq 3\ell_j/2$,
- (d) for every $j \in [1. t]$, if P_j is k -mismatch periodic with period Q_j , then $\text{hd}(P_{j+1}, Q_j^\infty) \leq 2k + 1$.

Proof. Properties (a), (b), and (c) are straightforward. For Property (d), there are two possible cases: if P_{j+1} is k -mismatch periodic, Claim 4.2 implies that P_{j+1} has the same k -mismatch period Q_j as P_j , that is $\text{hd}(P_{j+1}, Q_j^\infty) \leq 2k$. Otherwise, by construction, P_{j+1} is the shortest extension of P_j that is not k -mismatch periodic. By minimality, removing its last character yields a k -mismatch periodic prefix, and by Claim 4.2, it has the same k -mismatch period Q_j as P_j , i.e., we have $\text{hd}(P[. \ell_{j+1}), Q_j^\infty) \leq 2k$ for $i < \ell_j$. Adding one more character to $P[. \ell_{j+1})$ can increase the Hamming distance by at most one. ◁

Processing the pattern. In the preprocessing phase, we build \mathcal{P} and, for each k -mismatch periodic prefix $P_j \in \mathcal{P} \setminus \{P\}$, we also retrieve the period Q_j (represented as a fragment of P_j) and the mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$. For subsequent indices $j \in [0. \lceil \log_{3/2} m \rceil]$, we add the prefix R_j to \mathcal{P} . If $R_j \neq P$, we apply Corollary 4.4 to test whether R_j is k -mismatch periodic and, if so, retrieve the period Q . If R_j is k -mismatch periodic, we build $\text{MI}(R_j, Q^\infty)$ and extend R_j while maintaining the mismatch information with the appropriate prefix of Q^∞ . We proceed until we reach length $|R_{j+1}|$ or $2k + 1$ mismatches, whichever comes first. We add the obtained extension R'_j to \mathcal{P} and store the mismatch information $\text{MI}(R'_j, Q^\infty)$. If $\text{hd}(R'_j, Q^\infty) \leq 2k$, then $R'_j = R_{j+1}$ is k -mismatch periodic with the same period Q . Otherwise, by Claim 4.2, neither R'_j nor R_{j+1} are k -mismatch periodic. Processing each j takes $O(|R_{j+1}|k)$ time and $O(k)$ space, for a total of $O(mk)$ time and $O(k \log m)$ space across $j \in [0. \lceil \log_{3/2} m \rceil]$.

Processing the text. Our online algorithm processing the text T consists of $t = |\mathcal{P}|$ layers, each of which reports the k -mismatch occurrences of $P_j \in \mathcal{P}$, along with the underlying mismatch information.

The first layer, responsible for $P_1 = P[1]$, is implemented naively in $O(1)$ space and time per character.

Each of the subsequent layers receives the k -mismatch occurrences of P_j and outputs the k -mismatch occurrences of P_{j+1} . The processing is based on the following simple observation:

► **Observation 4.7.** *If P_{j+1} has a k -mismatch occurrence at position i of T , then P_j has a k -mismatch occurrence at position $i - \ell_{j+1} + \ell_j$ of T .*

10:12 Online Language Distance Problem for Palindromes and Squares

We partition T into blocks of length $b := \lceil \ell_j/2 \rceil$ and, for each block $T(rb..(r+1)b]$, use a separate subroutine to output k -mismatch occurrences of P_{j+1} at positions $i \in (rb..(r+1)b]$. This subroutine receives the k -mismatch occurrences of P_j at positions $i - \ell_{j+1} + \ell_j \in (rb - \ell_{j+1} + \ell_j..(r+1)b - \ell_{j+1} + \ell_j]$. It is considered *active* as the algorithm reads $T(rb - \ell_{j+1} + \ell_j..(r+1)b]$; since $\ell_{j+1} \leq \frac{3}{2}\ell_j$, at most two subroutines are active at any given time. The implementation of the subroutine depends on whether P_j is k -mismatch periodic or not.

P_j is not k -mismatch periodic. In this case, for every received k -mismatch occurrence i' of P_j , the subroutine stores the mismatch information $\text{MI}(T(i' - \ell_j..i'), P_j)$ and, as the algorithm receives subsequent characters $T[i]$ for $i \in (i'..i' + \ell_{j+1} - \ell_j]$, we maintain $\text{MI}(T(i' - \ell_j..i), P[.. \ell_j + i - i'])$ as long as there are at most k mismatches. If this is still the case for $i = i' + \ell_{j+1} - \ell_j$, we report a k -mismatch occurrence of P_{j+1} and output $\text{MI}(T(i' - \ell_j..i), P[.. \ell_j + i - i']) = \text{MI}(T(i - \ell_{j+1}..i), P_{j+1})$. By Observation 4.7, no k -mismatch occurrence of P_{j+1} is missed. Moreover, Fact 4.3 guarantees that the subroutine receives $O(k)$ k -mismatch occurrences of P_j , and thus it uses $O(k)$ space and $O(k)$ time per character.

P_j is k -mismatch periodic with period Q_j . In this case, we wait for the leftmost k -mismatch occurrence $p \in (rb - \ell_{j+1} + \ell_j..(r+1)b - \ell_{j+1} + \ell_j]$ of P_j and ignore all the subsequent occurrences of P_j . We use the received mismatch information $\text{MI}(T(p - \ell_j..p), P_j)$ and the preprocessed mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$ to construct $\text{MI}(T(p - \ell_j..p), Q_j^\infty)$; by the triangle inequality, the size of this set is guaranteed to be at most $3k$. As the algorithm receives subsequent characters of $T[i]$ for $i \in (p..(r+1)b]$, we maintain $\text{MI}(T(p - \ell_j..i), Q_j^\infty)$ as long as the number of mismatches does not exceed $6k + 1$. Whenever $i \geq p + \ell_{j+1} - \ell_j$ and $i \equiv p + \ell_{j+1} - \ell_j \pmod{|Q_j|}$, we extract $\text{MI}(T(i - \ell_{j+1}..i), Q_j^\infty)$ from $\text{MI}(T(p - \ell_j..i), Q_j^\infty)$ and use the precomputed mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$ to construct $\text{MI}(T(i - \ell_{j+1}..i), P_j)$. If it is of size at most k , we report i as a k -mismatch occurrence of P_j .

As for the correctness, we argue that we miss no k -mismatch occurrence $i \in (rb..(r+1)b]$ of P_{j+1} in T . Since $\text{hd}(T(i - \ell_{j+1}..i), P_{j+1}) \leq k$ and $\text{hd}(P_{j+1}, Q_j^\infty) \leq 2k + 1$, we have $\text{hd}(T(i - \ell_{j+1}..i), Q_j^\infty) \leq 3k + 1$. Moreover, by Observation 4.7, $i - \ell_{j+1} + \ell_j$ is a k -mismatch occurrence of P_j . Fact 4.3 further implies that $i - \ell_{j+1} + \ell_j \equiv p \pmod{|Q_j|}$ and $\text{hd}(T(p - \ell_j..i - \ell_{j+1}), Q_j^\infty) \leq 3k$. Consequently, $\text{hd}(T(p - \ell_j..i), Q_j^\infty) \leq 6k + 1$, and thus we compute $\text{MI}(T(i - \ell_{j+1}..i), Q_j^\infty)$ and report i as a k -mismatch occurrence of P_{j+1} .

We conclude with the complexity analysis: the working space is $O(k)$, dominated by the maintained mismatch information. Moreover, whenever we compute $\text{MI}(T(i - \ell_{j+1}..i), P_j)$, the size of this set is, by the triangle inequality, at most $6k + 1 + 2k + 1 \leq 8k + 2$, and it can be computed in $O(k)$ time.

Summary. Overall, each subroutine of each level takes $O(k)$ space and $O(k)$ time per character. Since there are $t = O(\log m)$ levels and each level contains at most two active subroutines, the algorithm takes $O(k \log m)$ space and $O(k \log m)$ time per text character. Although our pattern preprocessing algorithm is an offline procedure, we can run it while the algorithm reads the first $m/2$ characters of the text. Then, while the algorithm reads further $m/2$ characters, it can process two characters at a time to catch up with the input stream. This does not result in any delay on the output because the leftmost k -mismatch occurrence of P is at position m or larger.

4.3 Read-only algorithm for k -LHD-PAL

► **Theorem 4.8.** *There is a deterministic online algorithm that solves the k -LHD-PAL problem for a string of length n using $O(k \log n)$ space and $O(k \log n)$ worst-case time per character.*

The algorithm uses a filtering approach to select positions where a prefix close to PAL can end. Define a family $\mathcal{P} = \{P_j = T[.(3/2)^j] : j \in [1..\lceil \log_{3/2} n \rceil]\}$ of prefixes of the text, and let $\ell_j = |P_j|$, setting $\ell_0 = 0$ for notational convenience.

▷ **Claim 4.9.** Consider $j \in [1..\lceil \log_{3/2} n \rceil]$ and a position $i \in (2\ell_{j-1}..2\ell_j]$. If $\text{hd}(T[.i], \text{PAL}) \leq k$, then i is a $2k$ -mismatch occurrence of P_j^R in T . Moreover, $\text{hd}(T[.i], \text{PAL}) = \text{hd}(T(i - i'..i), P_j[1..i']^R)$ for $i' = \lfloor i/2 \rfloor$.

Proof. Note that $i > 2\ell_{j-1} \geq \ell_j$ implies that P_j is a prefix of $T[.i]$ and, equivalently, P_j^R is a suffix of $T[.i]^R$. Property 2.1 implies $2 \cdot \text{hd}(T[.i], \text{PAL}) = \text{hd}(T[.i], T[.i]^R) \geq \text{hd}(T(i - \ell_j..i), P_j)$. Thus, if $\text{hd}(T[.i], \text{PAL}) \leq k$, then i is a $2k$ -mismatch occurrence of P_j in T . Since $T[.i']$ is a prefix of P_j , Property 2.1 further implies $\text{hd}(T[.i], \text{PAL}) = \text{hd}(T(i - i'..i), T[.i']^R) = \text{hd}(T(i - i'..i), P_j[1..i']^R)$. ◁

The algorithm constructs the family \mathcal{P} as it reads the text. For each level j , we implement a subroutine responsible for positions $i \in (2\ell_{j-1}..2\ell_j]$. First, while reading $T[\ell_j..2\ell_{j-1}]$, we launch the pattern-matching algorithm of Theorem 4.5 in order to compute the $2k$ -mismatch occurrences of P_j^R in $T_j = T[.2\ell_j]$ and feed the pattern-matching algorithm with the pattern P_j and a prefix $T[.2\ell_{j-1}]$ of T_j , ignoring any output produced. The total number of characters provided is $\ell_j + 2\ell_{j-1} \leq 7 \cdot (2\ell_{j-1} - \ell_j)$, so we can feed the algorithm with $O(1)$ characters for every scanned character of T . Then, while reading $T[2\ell_{j-1}..2\ell_j]$, we feed the pattern-matching algorithm with subsequent characters of T . For every reported $2k$ -mismatch occurrence i of P_j^R in T_j , we retrieve the mismatch information $\text{MI}(T(i - \ell_j..i), P_j^R)$ and obtain $\text{MI}(T(i - i'..i), P_j[1..i']^R)$ by removing the entries corresponding to the leftmost $\ell_j - i'$ positions. We report the size of this set (or ∞ if the size exceeds k) as $\text{hd}_{\leq k}(T[.i], \text{PAL})$.

By Claim 4.9, all positions $i \in (2\ell_{j-1}..2\ell_j]$ such that $\text{hd}(T[.i], \text{PAL}) \leq k$ pass the test and the distance $\text{hd}(T[.i], \text{PAL})$ is equal to the size of the set $\text{MI}(T(i - i'..i), P_j[1..i']^R)$. As for the complexity analysis, observe that, for each level j , the pattern-matching algorithm uses $O(k \cdot j)$ space and takes $O(k \cdot j)$ time per character. Since, at any time, there is a constant number of active levels, the main algorithm uses $O(k \log n)$ space and takes $O(k \log n)$ time per character.

4.4 Read-only algorithm for k -LHD-SQ

► **Theorem 4.10.** *There is a deterministic online algorithm that solves the k -LHD-SQ problem for a string $T \in \Sigma^n$ using $O(k \log n)$ space and $O(k \log n)$ worst-case time per character.*

Our algorithm is very similar to the pattern-matching algorithm of Theorem 4.5. We use the same sequence $\mathcal{P} = (P_j)_{j=1}^t$ of prefixes, now defined for $P = T$. Again, we set $\ell_j = |P_j|$ for $j \in [1..t]$. Instead of Observation 4.7, we use Observation 3.4 to argue that our filtering procedure is correct.

Processing \mathcal{P} . We build \mathcal{P} in an online fashion so that the prefix P_j is constructed while scanning $T(\ell_j..\lceil 3\ell_j/2 \rceil)$. If P_j is k -mismatch periodic, then we also identify P_{j+1} and build $\text{MI}(P_{j+1}, Q_j^\infty)$.

For subsequent indices $j \in [0..[\log_{3/2} n]]$, we add the prefix R_j to \mathcal{P} as soon as it has been read. Then, we launch an offline procedure that applies Corollary 4.4 to test whether R_j is k -mismatch periodic and, if so, retrieves the period Q . If R_j is k -mismatch periodic, we build $\text{MI}(R_j, Q^\infty)$ and extend R_j while maintaining the mismatch information with the appropriate prefix of Q^∞ . We proceed until we reach length $|R_{j+1}|$ or $2k + 1$ mismatches, whichever comes first. We add the obtained extension R'_j to \mathcal{P} and store the mismatch information $\text{MI}(R'_j, Q^\infty)$. If $\text{hd}(R'_j, Q^\infty) \leq 2k$, then $R'_j = R_{j+1}$ is k -mismatch periodic with the same period Q . Otherwise, by Claim 4.2, neither R'_j nor R_{j+1} are k -mismatch periodic. Processing each j takes $O(|R_{j+1}|k)$ time and $O(k)$ space, and this computation needs to be completed while the algorithm reads $T(|R_j|.|R_{j+1}|)$. This gives $O(k)$ time per position since $\lceil \frac{3}{2}|R_j| \rceil \leq |R_{j+1}| \leq \lceil \frac{3}{2}|R_j| \rceil$.

Across all indices $j \in [0..[\log_{3/2} n]]$, the preprocessing algorithm takes $O(k)$ space and time per character (since no two indices are processed simultaneously).

Computing the distances. For each level $j \in [1..t]$, we implement a subroutine responsible for even positions $i \in [2\ell_j..2\ell_{j+1}]$; this procedure is active as we read $T[\ell_j..2\ell_{j+1}]$. As described above, the pattern P_j is identified while the algorithm reads $T[\ell_j..[\frac{3\ell_j}{2}]]$ and, if P_j is k -mismatch periodic, the period Q_j and the mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$ are also computed at that time. While reading $T[[\frac{3\ell_j}{2}]..2\ell_j)$, we launch the pattern-matching algorithm of Theorem 4.5 to report the k -mismatch occurrences of P_j in $T_j = T[.\ell_j + \ell_{j+1})$ and feed this algorithm with the pattern P_j and the prefix $T[.2\ell_j)$ of the text T_j . The total number of characters provided is $3\ell_j \leq 6 \cdot \frac{1}{2}\ell_j$, so can feed the pattern-matching algorithm with $O(1)$ character for every scanned character of T . Then, while reading $T[2\ell_j.. \ell_j + \ell_{j+1})$, we feed the pattern-matching algorithm subsequent text characters. For every $i' \in [2\ell_j.. \ell_j + \ell_{j+1})$, we learn whether i' is a k -mismatch occurrence of P_j and, if so, we obtain the mismatch information $\text{MI}(P_j, T(i' - \ell_j..i'))$. How we utilise this output depends on whether P_j is k -mismatch periodic or not: if P_j is not k -mismatch periodic, then T_j contains $O(k)$ k -mismatch occurrences of P_j and storing them explicitly requires little space. When P_j is k -mismatch periodic, T_j must exhibit similar periodicity, which we can use to avoid storing all occurrences explicitly.

P_j is not k -mismatch periodic. In this case, for every received k -mismatch occurrence i' of P_j , we store the mismatch information $\text{MI}(T(i' - \ell_j..i'), P_j)$ and, as the algorithm receives subsequent characters $T[i]$ for $i \in (i'..2(i' - \ell_j)]$, we maintain $\text{MI}(T(i' - \ell_j..i), T[.\ell_j + i - i'])$ as long as there are at most k mismatches. If this is still the case for $i = 2(i' - \ell_j)$, we report that $T[.i)$ is a k -mismatch square, with $\text{hd}(T[.i), \text{SQ}) = \text{hd}(T(i' - \ell_j..i), T[.\ell_j + i - i']) = \text{hd}(T(i/2..i), T[.i/2])$. By Observation 3.4, no k -mismatch square $T[.i)$ is missed. Moreover, Fact 4.3 guarantees that there are $O(k)$ k -mismatch occurrences of P_j , and thus we use $O(k)$ space and $O(k)$ time per character to process all of them.

P_j is k -mismatch periodic with period Q_j . In this case, we wait for the leftmost k -mismatch occurrence $p \in [2\ell_j.. \ell_j + \ell_{j+1})$ of P_j and ignore all the subsequent occurrences of P_j . We use the received mismatch information $\text{MI}(T(p - \ell_j..p), P_j)$ and the preprocessed mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$ to construct $\text{MI}(T(p - \ell_j..p), Q_j^\infty)$; by the triangle inequality, the size of this set is guaranteed to be at most $3k$. As the algorithm receives subsequent characters of $T[i]$ for $i \in (p..2\ell_{j+1})$, we maintain $\text{MI}(T(p - \ell_j..i), Q_j^\infty)$ as long as the number of mismatches does not exceed $6k + 1$. Whenever $i/2 \geq p - \ell_j$ and $i/2 \equiv p - \ell_j \pmod{|Q_j|}$, we extract $\text{MI}(T(i/2..i), Q_j^\infty)$ from $\text{MI}(T(p - \ell_j..i), Q_j^\infty)$ and use the precomputed

mismatch information $\text{MI}(P_{j+1}, Q_j^\infty)$ to construct $\text{MI}(T[.i/2], Q_j^\infty)$ first, and then derive $\text{MI}(T[.i/2], T(i/2..i))$. If the latter is of size at most k , we report $T[.i]$ as a k -mismatch square.

As for the correctness, we argue that we miss no k -mismatch square $T[.i]$ with $i \in (2\ell_j..2\ell_{j+1}]$. Since $\text{hd}(T(i/2..i), T[.i/2]) \leq k$ and $\text{hd}(P_{j+1}, Q_j^\infty) \leq 2k + 1$, as a corollary we obtain $\text{hd}(T(i/2..i), Q_j^\infty) \leq 3k + 1$. Moreover, by Observation 3.4, $i/2 + \ell_j$ is a k -mismatch occurrence of P_j . Fact 4.3 further implies that $i/2 + \ell_j \equiv p \pmod{|Q_j|}$ and $\text{hd}(T(p - \ell_j..i/2), Q_j^\infty) \leq 3k$. Consequently, $\text{hd}(T(p - \ell_j..i), Q_j^\infty) \leq 6k + 1$, and thus we compute $\text{MI}(T[.i/2], T(i/2..i))$ and report $T[.i]$ as a k -mismatch square.

We conclude with the complexity analysis: the working space is $O(k)$, dominated by the maintained mismatch information. Moreover, whenever we compute $\text{MI}(T[.i/2], T(i/2..i))$, the size of this set is, by the triangle inequality, at most $6k + 1 + 2k + 1 \leq 8k + 2$, and it can be computed in $O(k)$ time.

Summary. Overall, each level takes $O(k \log n)$ space and $O(k \log n)$ time per character, dominated by the pattern-matching algorithm of Theorem 4.5. However, since constantly many levels are processed at any given time, the entire algorithm still uses $O(k \log n)$ space and $O(k \log n)$ time per character.

5 Language Edit Distance problems

The *edit distance* between two strings U and V , denoted by $\text{ed}(U, V)$, is the minimum number of character insertions, deletions, and substitutions required to transform U into V . Similar to the Hamming distance, the edit distance from a string U to PAL and SQ can be expressed in terms of self-similarity of U . This allows us to use similar approaches as for the Language Hamming distance problems, with tools for the Hamming distance replaced with appropriate tools for the edit distance. Details for the proof of these theorems can be found in the full version of the paper, available on arXiv at <https://arxiv.org/abs/2309.14788>.

By replacing the Hamming distance sketch [11] with the edit distance sketch of Bhattacharya and Koucký [7].

► **Theorem 5.1.** *There is a randomised streaming algorithm that solves the k -LED-PAL problem for a string of length n using $\tilde{O}(k^2)$ bits of space and $\tilde{O}(k^2)$ time per character.*

Furthermore, the results of Bhattacharya and Koucký [7] show a reduction from the edit distance to the Hamming distance via locally consistent string decompositions, which allows reducing the k -LED-SQ problem to k -LHD-SQ, solved via Proposition 3.10:

► **Theorem 5.2.** *There is a randomised streaming algorithm that solves the k -LED-SQ problem for a string of length n using $\tilde{O}(k^2)$ bits of space and $\tilde{O}(k^2)$ time per character.*

Finally, by replacing the online read-only algorithm for finding the k -mismatch occurrences of a pattern in a text with an online read-only algorithm for finding k -error occurrences and the structural results for the Hamming distance with the structural results for the edit distance, we obtain algorithms for k -LED-PAL and k -LED-SQ:

► **Theorem 5.3.** *There is a deterministic online read-only algorithm that solves the k -LED-PAL problem for a string of length n using $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ time per character.*

► **Theorem 5.4.** *There is a deterministic online read-only algorithm that solves the k -LED-SQ problem for a string of length n using $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortised time per character.*

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the Current Clique Algorithms Are Optimal, so Is Valiant’s Parser. *SIAM Journal on Computing*, 47(6):2527–2555, 2018. doi:10.1137/16M1061771.
- 2 Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972. doi:10.1137/0201022.
- 3 Amihood Amir and Benny Porat. Approximate on-line palindrome recognition, and applications. In *Proc. of CPM 2014*, volume 8486 of *LNCS*, pages 21–29. Springer, 2014. doi:10.1007/978-3-319-07566-2_3.
- 4 Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In *Proc. of PODS 2016*, pages 477–488. ACM, 2016. doi:10.1145/2902251.2902304.
- 5 Djamel Belazzougui and Mathieu Raffinot. Approximate regular expression matching with multi-strings. *Journal of Discrete Algorithms*, 18:14–21, 2013. doi:10.1016/j.jda.2012.07.008.
- 6 Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome recognition in the streaming model. In *Proc. of STACS*, volume 25, pages 149–161, 2014. doi:10.4230/LIPIcs.STACS.2014.149.
- 7 Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In *Proc. of 55th STOC*, pages 219–232. ACM, 2023. doi:10.1145/3564246.3585239.
- 8 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. *SIAM Journal on Computing*, 48(2):481–512, 2019. doi:10.1137/17M112720X.
- 9 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. of 61st FOCS*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.
- 10 Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. Faster min-plus product for monotone instances. In *Proc. of 54th STOC*, pages 1529–1542. ACM, 2022. doi:10.1145/3519935.3520057.
- 11 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k -mismatch problem. In *Proc. of SODA 2019*, pages 1106–1125. SIAM, 2019. doi:10.1137/1.9781611975482.68.
- 12 Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for Dyck edit distance and RNA folding. In *Proc. of ICALP 2022*, volume 229 of *LIPIcs*, pages 49:1–49:20, 2022. doi:10.4230/LIPIcs.ICALP.2022.49.
- 13 Anita Dür. Improved bounds for rectangular monotone Min-Plus Product and applications. *Information Processing Letters*, 181:106358, 2023. doi:10.1016/j.ipl.2023.106358.
- 14 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.1090/S0002-9939-1965-0174934-9.
- 15 Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. An improved algorithm for the k -Dyck edit distance problem. In *Proc. of SODA 2022*, pages 3650–3669. SIAM, 2022. doi:10.1137/1.9781611977073.144.
- 16 Zvi Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proc. of STOC*, pages 161–173. ACM, 1976. doi:10.1145/800113.803644.

- 17 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986. doi:10.1145/8307.8309.
- 18 Paweł Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemysław Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. doi:10.1007/s00453-019-00591-8.
- 19 Wei Huang, Yaoyun Shi, Shengyu Zhang, and Yufan Zhu. The communication complexity of the Hamming distance problem. *Information Processing Letters*, 99(4):149–153, 2006.
- 20 Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. Small-space and streaming pattern matching with k edits. In *Proc. of FOCS 2021*, pages 885–896. IEEE, 2021. doi:10.1109/FOCS52979.2021.00090.
- 21 Roman Kolpakov and Gregory Kucherov. Finding approximate repetitions under Hamming distance. *Theoretical Computer Science*, 303(1):135–156, 2003. Logic and Complexity in Computer Science. doi:10.1016/S0304-3975(02)00448-6.
- 22 Michal Koucký and Michael E. Saks. Simple, deterministic, fast (but weak) approximations to edit distance and Dyck edit distance. In *Proc. of SODA 2023*, pages 5203–5219. SIAM, 2023. doi:10.1137/1.9781611977554.ch188.
- 23 Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *Proc. of MFCS 2011*, volume 6907 of *LNCS*, pages 412–423. Springer, 2011. doi:10.1007/978-3-642-22993-0_38.
- 24 Gad M. Landau and Jeanette P. Schmidt. An algorithm for approximate tandem repeats. In *Proc. of CPM*, pages 120–133, 1993. doi:10.1007/BFb0029801.
- 25 Lillian Lee. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15, January 2002. doi:10.1145/505241.505242.
- 26 Oleg Merkurev and Arseny M. Shur. Computing the maximum exponent in a stream. *Algorithmica*, 84(3):742–756, 2022. doi:10.1007/s00453-021-00883-y.
- 27 Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995. doi:10.1016/0020-0190(95)00007-y.
- 28 Alexandre H. L. Porto and Valmir Carneiro Barbosa. Finding approximate palindromes in strings. *Pattern Recognit.*, 35(11):2581–2591, 2002. doi:10.1016/S0031-3203(01)00179-0.
- 29 Walter L. Ruzzo. On the complexity of general context-free language parsing and recognition. In *Proc. of ICALP 1979*, volume 71 of *LNCS*, pages 489–497. Springer, 1979. doi:10.1007/3-540-09510-1_39.
- 30 Wojciech Rytter. On maximal suffixes and constant-space linear-time versions of KMP algorithm. *Theoretical Computer Science*, 299(1-3):763–774, 2003. doi:10.1016/S0304-3975(02)00590-X.
- 31 Barna Saha. The Dyck language edit distance problem in near-linear time. In *Proc. of FOCS 2014*, pages 611–620. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.71.
- 32 Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *Proc. of FOCS 2015*, pages 118–135. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.17.
- 33 Barna Saha. Fast space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In *Proc. of FOCS 2017*, pages 295–306. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.35.
- 34 Giorgio Satta. Tree-adjointing grammar parsing and boolean matrix multiplication. *Comput. Linguistics*, 20(2):173–191, 1994. URL: <https://aclanthology.org/J94-2002>.
- 35 Dina Sokol, Gary Benson, and Justin Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):e30–e35, January 2007. doi:10.1093/bioinformatics/bt1309.
- 36 Dina Sokol and Justin Tojeira. Speeding up the detection of tandem repeats over the edit distance. *Theoretical Computer Science*, 525:103–110, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.04.021.