Streaming Periodicity with Mismatches, Wildcards, and Edits

Taha El Ghazi
DIENS, École normale supérieure de Paris, PSL Research University, France

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

Abstract

In this work, we study the problem of detecting periodic trends in strings. While detecting exact periodicity has been studied extensively, real-world data is often noisy, where small deviations or mismatches occur between repetitions. This work focuses on a generalized approach to period detection that efficiently handles noise. Given a string S of length n, the task is to identify integers p such that the prefix and the suffix of S, each of length $n - p + 1$, are similar under a given distance measure. Ergün et al. [APPROX-RANDOM 2017] were the first to study this problem in the streaming model under the Hamming distance. In this work, we combine, in a non-trivial way, the Hamming distance sketch of Clifford et al. [SODA 2019] and the structural description of the k-mismatch occurrences of a pattern in a text by Charalampopoulos et al. [FOCS 2020] to present a more efficient streaming algorithm for period detection under the Hamming distance. As a corollary, we derive a streaming algorithm for detecting periods of strings which may contain wildcards, a special symbol that match any character of the alphabet. Our algorithm is not only more efficient than that of Ergün et al. [TCS 2020], but it also operates without their assumption that the string must be free of wildcards in its final characters. Additionally, we introduce the first two-pass streaming algorithm for computing periods under the edit distance by leveraging and extending the Bhattacharya–Koucký's grammar decomposition technique [STOC 2023].

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases approximate periods, pattern matching, streaming algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2025.36

Related Version *Full Version*: https://arxiv.org/abs/2509.14898

Funding Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR) and a Royal Society International Exchanges Award.

1 Introduction

In this work, we consider the problem of computing periodic trends in strings. Informally, a string has period p if its prefix of length $n - p + 1$ equals its suffix of length $n - p + 1$. Alternatively, a string has period p if it equals its prefix of length p repeated a (possibly, fractional) number of times. The problem of detecting periodic trends in strings has numerous practical applications, including fields like astronomy, financial analytics, and meteorology (see e.g. [11]). The nature and the volume of the data in the applications ask for algorithms able to process the input in very little space and in (almost) real time, such as streaming algorithms. In the streaming setting, we assume that the input string arrives one character at a time and account for all space used, even for the space used to store information about the input data, which results in ultra-efficient algorithms.

The first streaming algorithm for detecting exact periods of a stream of length n was given by Ergün, Jowhari, and Sağlam [14], who presented an algorithm with $O(\log^2 n)$ space.[1]

[1] Hereafter, the space is measured in bits.

While the problem of detecting exact periodicity is fundamental to string processing, in practice data is rarely exactly periodic. For example, a string *abcabcabaabaaba* is clearly repetitive, but does not have a small exact period in the sense above. To account for these variations in natural data, Ergün, Jowhari, and Sağlam [14] proposed another streaming algorithm, which allows for approximately computing the Hamming distance (the number of mismatches) between a stream and a string with period p, for a given integer p. Their algorithm computes the distance with an approximation factor $2 + \varepsilon$ in space $\tilde{O}(1/\varepsilon^2)$. A disadvantage of this algorithm is that p must be known in advance, and knowing p approximately will not suffice.

Later, Ergün, Grigorescu, Azer, and Zhou [12] suggested a different version of this problem: given a string of length n and an integer k, they asked for all integer p such that the Hamming distance between the string and its copy shifted by p is at most k (see Section 2 for a definition). They called such p *k-mismatch periods*. They proved that for computing all k-mismatch periods of a streaming string of length n one needs $\Omega(n)$ space and that for computing all k-mismatch periods in $[1..n/2]$, one needs $\Omega(k \log n)$ space. On the other hand, prior work [12, 15] claimed a streaming algorithm that, given a string of length n, computes all its k-mismatch periods in $[1..n/2]$ using only $O(k^4 \log^9 n)$ space. However, we found significant gaps in the proof of correctness. Many key arguments are only briefly outlined in both the conference version [12] and the longer arXiv manuscript [15], and as confirmed in private communication with the authors, a full journal version of the paper has not been published. In particular, [15, Theorem 28] is proved only for the case when the string has two distinct k-mismatch periods, and is stated to generalize easily to the case when the string has more than two distinct periods. We believe this generalization does not hold; see Appendix A for a detailed explanation.

Finally, in a different work [13] Ergün, Grigorescu, Azer, and Zhou considered the problem of computing periods of incomplete streaming data, where some characters are replaced with wildcards, a special symbol that matches each character of the alphabet. We say that a wildcard-containing string T has an integer period p if T shifted by p positions matches itself. They showed that a streaming algorithm which computes all periods of an n-length string with wildcards requires $\Omega(n)$ space. Conversely, they demonstrated a streaming algorithm which given a string T with k wildcards computes all its periods $p \leq n/2$ under condition that there are no wildcards in the last p characters of T. The algorithm uses $O(k^3 \operatorname{polylog} n)$ space. They also showed that for $k = o(\sqrt{n})$, such an algorithm must use $\Omega(k \log n)$ space.

Our results

In this work, we continue the study initiated by Ergün, Grigorescu, Azer, and Zhou [12]. We first give a streaming algorithm for computing the k-mismatch periods of a string (Theorem 2.10), providing full details and improving over the claimed space bound of Ergün, Grigorescu, Azer, and Zhou [12] by a factor of $k^2 \log^5 n$.

As an almost immediate corollary, and our second contribution, we obtain a similar result for computing periods of a string with few wildcards (Theorem 2.2). Specifically, we both improve the complexity of the algorithm by Ergün, Grigorescu, Azer, and Zhou [13] by a factor of $k \operatorname{polylog} n$, narrowing the gap between the upper and lower bounds, and remove the assumption that there are no wildcards in the last characters of the input string.

As our third and final contribution, we extend our results to the edit distance. Informally, we say that a string has a *k-edit period* $p \leq n/2$ if the edit distance (the number of deletions, insertions, and substitutions required to transform one string into another) between the input string and its copy shifted by p characters is at most k (see Section 2 for a formal

definition). We apply the Bhattacharya–Koucký's grammar decomposition [4] and then show that we can compute the k-edit periods of the input by computing the k-mismatch periods of the stream of suitably encoded grammars. As our algorithm for k-mismatch periods needs to know the length of the stream (assumption already present in [12]), we make two passes: first, we compute the decomposition and the length of the stream, and in the second pass we compute the periods. As a result, we develop the first two-pass streaming algorithm for computing k-edit periods of a string in $\tilde{O}(k^2 n)$ time and $\tilde{O}(k^4)$ space (Theorem 2.3).[2] We emphasize that in this work, we deliberately chose to treat our algorithm for k-mismatch periods as a black box, using it as a direct application in the context of k-edit periods. While it is conceivable that opening this black box and combining its internal components with the Bhattacharya–Koucký technique could lead to a one-pass streaming algorithm for k-edit periods, pursuing this direction would require significant new insights. We leave this as a promising open question for future research.

Related work

Other repetitive string structures that have been considered in the literature include palindromes (strings that read the same forward and backwards) and squares (strings equal to two copies of a string concatenated together). Berebrink et al. [3] followed by Gawrychowski et al. [16] studied the question of computing the length of a maximal palindromic substring of a stream that is a palindrome. Merkurev and Shur [21] considered a similar question for squares. Bathie et al. [1] considered the problem of recognising prefixes of a streaming string that are at Hamming (edit) distance at most k from palindromes and squares.

2 Results and technical overview

In this work, we assume the word-RAM model of computation and work in a particularly restrictive streaming setting. In this setting, we assume that the input string arrives as a stream, one character at a time. We must account for all the space used, including the space used to store information about the input. Throughout, the space is measured in bits.

Notations and terminology

We assume to be given an alphabet Σ, the elements of which, called *characters*, can be stored in a single machine word of the word-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 $S^0 = \varepsilon$. Furthermore, S^∞ denotes an infinite string obtained by concatenating infinitely many copies of S. For a string $T \in \Sigma^n$ and $i \in [1..n]$,[3] the ith character of T is denoted by $T[i]$. We use $|T| = n$ to denote the length of T. For $1 \leq i, j \leq n$, $T[i..j]$ 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 them, i.e., we write $T[..j] = T[1..j]$ and $T[i..] = T[i..n]$. 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 $P = T[i..]$. A non-empty string $T \in \Sigma^n$ is *primitive* if $T^2[i..j] = T$ implies $i = 1$ or $i = |T| + 1$.

[2] Hereafter, $\tilde{O}$ means that we hide factors polylogarithmic in n. "Two-pass" means that the algorithm reads T as a stream twice.

[3] 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 $(i..j] = \{k \in \mathbb{Z} : i < k \leq j\}$.

2.1 k-mismatch periods

The Hamming distance between two strings S, T (denoted $\mathsf{hd}(S,T)$) 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, $\mathsf{MI}(S,T)$ as the set $\{(i, S[i], T[i]) : i \in [1..n] \text{ and } S[i] \neq T[i]\}$. An integer p is called a *k-mismatch period* of a string T of length n if $1 \leq p \leq n$ and $\mathsf{hd}(T[1..n-p+1], T[p..n]) \leq k$.

▶ **Theorem 2.1** (Informal statement of Theorem 2.10)**.** *Given a string T of length n and an integer k, there is a streaming algorithm that computes all k-mismatch periods of T in $[1..n/2]$. Furthermore, for each detected k-mismatch period p, it also returns $\mathsf{MI}(T[p..], T[..n-p+1])$. The algorithm uses $O(n \cdot k \log^5 n)$ time and $O(k^2 \log^3 n)$ space, and is correct w.h.p.*[4]

To develop our result, we start with a simple idea already present in [12]: If p is a k-mismatch period of T, then for all $1 \leq \ell \leq n-p$, the position p is a starting position of a k-mismatch occurrence of $T[1..\ell+1]$. This already allows to filter out candidate periods. To test whether an integer p is a k-mismatch period, we check if the Hamming distance between $T[1..n-p+1]$ and $T[p..n]$ is at most k. For this, we aim to use the Hamming distance sketches introduced by Clifford, Kociumaka, and Porat [9] for strings $T[1..n-p+1]$ and $T[p..n]$. There are two main challenges: First, we might discover a candidate k-mismatch period p after passing position $n-p+1$, which would make it impossible to compute the Hamming distance sketch of $T[1..n-p+1]$ due to streaming access limitations. To address this, as in [12], we divide the interval $[1..n/2]$ into a logarithmic number of subintervals, filtering positions in each subinterval using progressively shorter prefixes. The second challenge is that the number of candidate periods can be large, and we cannot store all the sketches. Here, we diverge significantly from [12]: To store the sketches in small space, we leverage the concatenability of the Hamming distance sketches of [9] and the structural regularity of k-mismatch occurrences as shown by Charalampopoulos, Kociumaka, and Wellnitz [8]. Combining the two ideas together is non-trivial and this is where the main novelty of our result for the Hamming distance is. The proof of Theorem 2.10 is given in Section 3.

Assume to be given a string containing wildcards. By replacing wildcards in a string with a new character, we immediately derive the following:

▶ **Theorem 2.2.** *Given a string T of length n containing at most k wildcards. There is a streaming algorithm that computes the set of periods of T in $[1..n/2]$ in $O(n \cdot k \log^5 n)$ time and $O(k^2 \log^3 n)$ space. The algorithm is correct w.h.p.*

Proof. If we replace the wildcards in T with a new character $\# \notin \Sigma$, then a period p of T is a k-mismatch period of the resulting string in the alphabet $\Sigma \cup \{\#\}$, and we can find it via Theorem 2.1. To check that the positions returned by the algorithm are periods of the original string, for each detected k-mismatch period, we retrieve the relevant mismatch information and verify that for all mismatching pairs of characters, at least one of those is the special character $\#$. ◀

2.2 k-edit periods

The edit distance between two strings S, T (denoted by $\mathsf{ed}(S,T)$) is the minimum number of character insertions, deletions, and substitutions required to transform S into T. We say that an integer p is a *k-edit period* of a string T of length n if $1 \leq p \leq |T|$ and for some $1 \leq i \leq n$, $\mathsf{ed}(T[1..i], T[p..n]) \leq k$.

[4] Hereafter, w.h.p. stands for probability at least $1 - 1/n^c$, for a constant $c > 1$.

▶ **Theorem 2.3.** *Given a string T of length n and an integer k, there is a two-pass streaming algorithm that computes all k-edit periods of T in $\tilde{O}(k^2 n)$ time and $\tilde{O}(k^4)$ space. The algorithm is correct w.h.p.*

2.2.1 Preliminaries: Grammar decomposition

We assume familiarity with the notion of straight-line programs (SLP) [18], which represent a subclass of context-free grammars. The size of an SLP G is the number of non-terminals and is denoted by $|G|$. Furthermore, G represents a unique string, called the *expansion* of G, $\mathsf{exp}(G)$. The length of $|\mathsf{exp}(G)|$ can be computed in $O(|G|)$ time and space [20].

▶ **Fact 2.4.** *Let G_X and G_Y be SLPs representing strings X and Y respectively, and $m = |G_X| + |G_Y|$. Both of the following hold:*

- *We can compute $d := \mathsf{ed}(X, Y)$ in $O((m + d^2)\log|XY|)$ time and $O(m \log|XY|)$ space (see [5, Proposition 2.1]).*
- *Given an integer k, we can find all $1 \le i \le |Y|$ such that $\mathsf{ed}(X, Y[i..]) \le k$ and the corresponding edit distances in $O((m + k^2)\log|XY|)$ time and $O(m \log|XY|)$ space.*

Proof. By [19] (see also the remark at the end of [10, Section 5]), all such positions i can be found in $(k+1)^2$ *longest common extension* (LCE) queries on a string Z equal to the reverse of XY. Given two positions $1 \le i, j \le |Z|$, an LCE query asks for the maximal ℓ such that $Z[i..i+\ell-1] = Z[j..j+\ell-1]$. The string Z can be represented as the expansion of an SLP of size $O(|G_X| + |G_Y|)$. I [17] showed that after $O(m \log|Z|)$-time and -space preprocessing of the SLP[5], LCE queries on Z can be answered in $O(\log|Z|)$ time. The claim follows. ◀

One of the central tools of our solution is Bhattacharya–Koucký's grammar decomposition (BK-decomposition) [4]. It is a randomised decomposition that uses as source of randomness two families of hash functions $C_1, \ldots, C_L$ and $H_0, \ldots, H_L$, where $L = O(\log n)$ is a suitably chosen parameter. The decomposition of a string X is a sequence $\mathcal{G}(X)$ of SLPs.[6] For a sequence of SLPs $\mathcal{G} = G_1 \cdots G_m$, define $\mathsf{exp}(\mathcal{G}) = \mathsf{exp}(G_1) \cdots \mathsf{exp}(G_m)$.

▶ **Fact 2.5** ([4, Theorem 3.1])**.** *Let $X \in \Sigma^{\le n}$, $k \in \mathbb{N}$ be the input parameter of the grammar decomposition algorithm, and let $\mathcal{G}(X) = G_1^X \cdots G_s^X$. For all n large enough, $X = \mathsf{exp}(\mathcal{G}(X))$ and $|G_i^X| = \tilde{O}(k)$ for $i \in \{1, \ldots, s\}$ with probability $\ge 1 - 2/n$.*

The BK-decomposition can be maintained in a streaming fashion:

▶ **Corollary 2.6** ([6, Lemma 4.2,Theorem 5.1])**.** *Given a streaming string $X \in \Sigma^*$, there is an algorithm that outputs a stream of SLPs $\mathcal{G}_{\text{def}} = G_1 \cdots G_s$ (referred to as* definite*) and maintains a sequence $\mathcal{G}_{\text{active}} = G'_1 \cdots G'_t$ of $\tilde{O}(1)$ SLPs (referred to as* active*) such that after having read $X[1..i]$, the concatenation of $\mathcal{G}_{\text{def}}$ and $\mathcal{G}_{\text{active}}$ equals $\mathcal{G}(X[1..i])$. The algorithm uses $\tilde{O}(k)$ time per character and $\tilde{O}(k)$ space (here, we account for all the space used except for the space required to store $\mathcal{G}_{\text{def}}$). The only operation the algorithm is allowed to perform on $\mathcal{G}_{\text{def}}$ is appending a grammar from the right.*

We further make use of the following claim:

[5] The result of I [17] gives tighter bounds, but this is sufficient for our purposes

[6] Strictly speaking, the decomposition [4] outputs run-length SLPs. However, one can transform those grammars into SLPs with a size blow-up polylogarithmic in n [20].

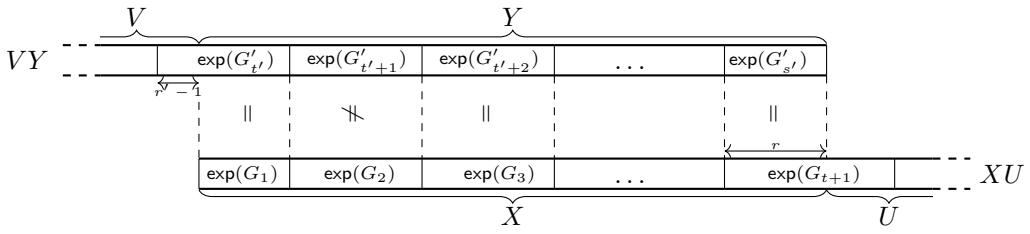

Figure 1 Grammar decompositions for VY and XU for the case $\mathsf{ed}(X,Y) \leq k$.

▶ **Corollary 2.7.** *Let $k \leq n$ be integers. Assume $X,Y,U,V \in \Sigma^*$, $|XU|,|VY| \leq n$, and $\mathsf{ed}(X,Y) \leq k$. Let $\mathcal{G}(XU) = G_1 \cdots G_s$ and $\mathcal{G}(VY) = G'_1 \cdots G'_{s'}$. With probability at least $1-1/5$, there exist integers r, r', t, t' such that each of the following is satisfied (see Figure 1):*

1. $t+1 = s'-t'+1$.
2. $X = \mathsf{exp}(G_1 \cdots G_t) \cdot \mathsf{exp}(G_{t+1})[..r]$ *and* $Y = \mathsf{exp}(G'_{t'})[r'..] \cdot \mathsf{exp}(G'_{t'+1} \cdots G'_{s'})$.
3. $G_i = G'_{t'+i-1}$ *except for at most $k+2$ indices $1 \leq i \leq t+1$.*
4. $\mathsf{ed}(X,Y)$ *equals the sum of* $\mathsf{ed}(\mathsf{exp}(G_1), \mathsf{exp}(G'_{t'})[r'..])$, $\sum_{2 \leq i \leq t} \mathsf{ed}(\mathsf{exp}(G_i), \mathsf{exp}(G'_{t'+i-1}))$, *and* $\mathsf{ed}(\mathsf{exp}(G_{t+1})[..r], \mathsf{exp}(G'_{s'}))$.

Bhattacharya and Koucký [4] proved a similar result in the case where V' and V are appended to X and Y, respectively, from the left. By construction, the BK-decomposition is (almost) symmetric, which allows to adapt their argument to show an analogous result for the case where U and U' are appended to X and Y, respectively, from the right. Corollary 2.7 follows by first applying the result of Bhattacharya and Koucký [4] with $V' = \varepsilon$, and then using our analogous argument with $U' = \varepsilon$.

Finally, we use the following encoding of SLPs that allows reusing algorithms on strings for sequences of SLPs:

▶ **Fact 2.8** ([4, Lemma 3.13])**.** *Let $\mu = \tilde{O}(k)$ be a parameter. There is a mapping* enc *from the set of SLPs output by the BK-decomposition algorithm to the set of strings of length μ on an alphabet Γ of size polynomial in n (the maximum length of the input string of the BK-decomposition algorithm) that guarantees that the following is satisfied:*

1. *A grammar can be encoded and decoded in $O(\mu)$ time and space;*
2. *Encodings of two equal grammars are equal;*
3. *Encodings of two distinct grammars output by the decomposition algorithm differ in all μ characters with probability at least $1-1/n$.*

2.2.2 Proof of Theorem 2.3

A high-level idea of our algorithm is to apply the grammar decomposition to the stream, and then to reduce the problem to the problem of computing periods with mismatches via Corollary 2.7 and Fact 2.8. To this end, we need a stronger version of a streaming algorithm for computing periods with mismatches, and in particular, we introduce a *weight* function w on strings. In the case of the Hamming distance, this function is identically zero. In case of the edit distance, we define it as a partial function on Γ^*, namely, if $\Gamma^* \ni S = \mathrm{enc}(G_1 \cdots G_m)$, then $w(S) = |\mathsf{exp}(G_1 \cdots G_m)|$.

▶ **Proposition 2.9.** *The weight functions defined above satisfy each of the following:*

- *For a string S of length n, $w(S)$ can be computed in streaming deterministically using $t_w(n)$ time per character and $s_w(n)$ space;*
- *Consider strings X, Y. If at least two of $w(X), w(Y), w(XY)$ are defined, then the third one is defined as well and $w(XY) = w(X) + w(Y)$;*
- *Consider strings X, Y that have equal length in $[1..n]$. If $w(X), w(Y)$ are defined, then $w(Y)$ can be computed deterministically in $|\mathsf{MI}(X, Y)| \cdot t_w(n)$ time and $s_w(n)$ space.*

For the Hamming distance, $t_w(n) = O(1)$ and $s_w(n) = O(1)$ (trivially), and for the edit distance $t_w(n) = \tilde{O}(1)$ and $s_w(n) = \tilde{O}(k)$.

Proof. The claim is trivial for $w \equiv 0$. In the following, we consider the weight function for the edit distance. First, consider $S = \mathrm{enc}(G_1 G_2 \cdots G_m)$. Its weight $w(S)$ can be computed in streaming using $t_w(n)$ time per character, where $t_w(n) = \tilde{O}(1)$ and $s_w(n) = \tilde{O}(k)$ space thanks to Fact 2.8. The second property obviously holds. Finally, assume $X = \mathrm{enc}(G_1 G_2 \dots G_m)$ and $Y = \mathrm{enc}(G'_1 G'_2 \dots G'_m)$. Let $\mathcal{I} = \{i \le m, \mathrm{enc}(G_i) \neq \mathrm{enc}(G'_i)\}$. Given $\mathsf{MI}(X, Y)$, because of Fact 2.8, we have access to all characters in $\mathrm{enc}(G_i)$ and $\mathrm{enc}(G'_i)$ for $i \in \mathcal{I}$. As a result, we can compute $w(Y) = w(X) + \sum_{i \in \mathcal{I}} w(\mathrm{enc}(G'_i)) - w(\mathrm{enc}(G_i))$ in $\mathsf{MI}(X, Y) \cdot t_w(n)$ time and $s_w(n)$ space. ◀

In Section 3, we show the following theorem:

▶ **Theorem 2.10.** *Given a string T of length n and integers k, Δ, there is a streaming algorithm that computes all k-mismatch periods of T in $[1..n/2 + \Delta]$. Furthermore, for each detected k-mismatch period p, it also returns weight $w(T[p..])$ and $\mathsf{MI}(T[p..], T[..n - p + 1])$. The algorithm runs in $O(n \cdot k t_w(n) \log^5 n)$ time, uses $O(k^2 \log^3 n + s_w(n) + \Delta \cdot (k \log n + s_w(n)))$ space, and is correct w.h.p., where $t_w(n)$ and $s_w(n)$ are as defined in Proposition 2.9.*[7]

Let us now explain how it implies the algorithm for computing k-edit periods. Recall that we receive a string T of length n as a stream. In the first streaming pass on T, we apply Corollary 2.6. At every moment of the algorithm, we additionally store the number of definite grammars. By Corollary 2.6, we can then compute $m := |\mathcal{G}(T)|$ in $\tilde{O}(kn)$ time and $\tilde{O}(k)$ space.

In the second streaming pass, we again run the algorithm of Corollary 2.6. When a new character arrives, we update the SLPs and if a SLP G becomes definite, i.e. if we append G to the stream of definite grammars, we compute $\mathrm{enc}(G)$ in $\tilde{O}(k)$ time and space (Fact 2.8), and feed it character-by-character into the $\mu(k+2)$-mismatch period algorithm (Theorem 2.10), where μ is the parameter from Fact 2.8. Consider the moment when we arrive at the end of T and let $G_1, \dots, G_t$ be the active grammars. We compute $\mathrm{enc}(G_1 \cdots G_t)$ and feed it by character-by-character into the $\mu(k+2)$-mismatch period algorithm. Let $\mathcal{T}$ be the resulting stream we fed into the algorithm, i.e. $\mathcal{T} = \mathrm{enc}(\mathcal{G}(T))$, where $\mathcal{G}(T) = G_1^T G_2^T \cdots G_m^T$.

The $\mu(k+2)$-mismatch period algorithm retrieves all $\mu(k+2)$-mismatch periods of $\mathcal{T}$ one-by-one. When a $\mu(k+2)$-mismatch period p is retrieved, we do the following. If $p - 1$ is not a multiple of μ, we discard it immediately. Otherwise, by Corollary 2.7, there are at most $k + 2$ values $2 \le i \le m - (p-1)/\mu - 1$ such that $\mathrm{enc}(G_i^T) \neq \mathrm{enc}(G^T_{i+(p-1)/\mu})$. We retrieve the set I containing all such i from $\mathsf{MI}(\mathcal{T}[p..], \mathcal{T}[..m \cdot \mu - p + 1])$. We finally compute

$$d(p) = \sum_{i \in I} \mathsf{ed}(\mathsf{exp}(G_i^T), \mathsf{exp}(G^T_{i+(p-1)/\mu})) + \min_r \mathsf{ed}(\mathsf{exp}(G^T_{m-(p-1)/\mu})[..r], \mathsf{exp}(G_m^T))$$

[7] By taking $w \equiv 0$ and $\Delta = 0$, we immediately obtain Theorem 2.1.

If $d(p) \le k$, we compute all r' such that $\mathsf{ed}(\mathsf{exp}(G_1^T), \mathsf{exp}(G^T_{1+(p-1)/\mu}[r'..])) \le k - d(p)$. For all such r', we return $w(\mathrm{enc}(G^T_{(p-1)/\mu+2} \dots G^T_m)) + r'$ as a k-edit period of T.

Correctness. If $p \le n/2$ is a k-edit period of T, there exists i such that $\mathsf{ed}(T[..i], T[p..]) \le k$. By Corollary 2.7, there exist integers r, r', t, t' such that $T[..i] = \mathsf{exp}(G_1^T \cdots G_t^T) \cdot \mathsf{exp}(G^T_{t+1})[..r]$, $T[p..] = \mathsf{exp}(G_{t'})[r'..] \cdot \mathsf{exp}(G^T_{t'+1} \cdots G^T_m)$, $t + 1 = m - t' + 1$ and there are at most $k + 2$ indices i such that G_i^T and $G^T_{t'+i}$ mismatch. By Fact 2.8, $t' \cdot \mu + 1$ is a $\mu \cdot (k + 2)$-mismatch period of $\mathcal{T}$. Further, $t' \le (m + k)/2 + 1$:

▷ Claim 2.11. $t' \le (m + k)/2 + 1$.

Proof. We have $i \ge (n - p + 1) - k$ and hence $i + (n - p + 1) \ge n - k$. Consequently, $t+1+(m-t'+1) \ge m-k$. Indeed, assume towards a contradiction that $t+(m-t'+1) \le m-k$. We then have

$$
\begin{aligned}
&i + (n - p + 1) \le \\
&|\mathsf{exp}(G_1^T \cdots G_t^T) \cdot \mathsf{exp}(G^T_{t+1})[..r]| + |\mathsf{exp}(G_{t'})[r'..] \cdot \mathsf{exp}(G_{t'+1} \cdots G_m)| \le \\
&|\mathsf{exp}(G_1^T \cdots G^T_{t+1})| + |\mathsf{exp}(G^T_{t'} \cdots G^T_m)| \le |\mathsf{exp}(G_1^T \cdots G^T_{t+1})| + |\mathsf{exp}(G^T_{t+k+2} \cdots G^T_m)| \le \\
&n - k
\end{aligned}
$$

where the last inequality holds as each SLP G_i^T, $t + 1 < i < t + k + 2$ is non-trivial and hence its expansion has length at least one. Finally, since $t + 1 = m - t' + 1$, we obtain $m - t' + 1 \ge (m - k)/2$ and therefore $t' \le (m + k)/2 + 1$. ◁

Consequently, $t' \cdot \mu + 1$ is detected by the $\mu \cdot (k + 2)$-mismatch period algorithm with $\Delta = \mu(k/2 + 1) + 1$. We then identify p as a k-edit period of T by computing the edit distances between the expansions of the mismatching SLPs.

The algorithm fails if Corollary 2.7 fails, or if the $\mu \cdot (k + 2)$-mismatch period algorithm fails, or if Fact 2.8 fails, which happens with probability $\le \frac{2}{5}$ for n big enough. Also, notice that when Corollary 2.7 fails, the algorithm computes edit distance that is bigger than the actual distance. As a result, with a standard argument, we can run $\log n$ instances of the algorithm and return the minimal computed distance, to have the algorithm succeed w.h.p.

Complexity. The first pass on T takes $\tilde{O}(kn)$ time and $\tilde{O}(k)$ space. For the second pass, the $\mu \cdot (k + 2)$-mismatch period algorithm takes $\tilde{O}(\mu k t_w(n) n) = \tilde{O}(k^2 n)$ time and $\tilde{O}((\mu k)^2 + s_w(n)) = \tilde{O}(k^4)$ space by Theorem 2.10 and Proposition 2.9. For each $\mu \cdot (k + 2)$-mismatch period returned by the algorithm, we can bound the time needed to compute $d(p)$ as follows: Let $k_i = \mathsf{ed}(\mathsf{exp}(G_i), \mathsf{exp}(G_{i+(p-1)/\mu}))$. By Fact 2.4, we can compute k_i in time $\tilde{O}(k_i^2)$, hence if $d(p) \le k$, we can compute $d(p)$ in time $\tilde{O}(\sum_i k_i^2) = \tilde{O}(k^2)$. If the edit distance computations take total time more than $\tilde{O}(k^2)$, we can terminate them as we know that $d(p) > k$.[8] Theorem 2.3 follows.

3 Proof of Theorem 2.10

In this section, we prove Theorem 2.10. We first recall essential tools for the Hamming distance, then outline our algorithm and solve each case, and finally analyse it.

[8] To be more precise, we upper bound the number of longest common extension queries from Fact 2.4, it should not exceed $\sum_i (k_i + 1)^2 \le (k + 1)^2$.

3.1 Tools for the Hamming distance

For two strings P, T, a position $i \in [|P|..|T|]$ of T is a *k-mismatch occurrence* of P in T if $\mathsf{hd}(T[i-|P|+1..i], P) \leq k$. For an integer k, we define $\mathsf{hd}_{\leq k}(X,Y) = \mathsf{hd}(X,Y)$ if $\mathsf{hd}(X,Y) \leq k$ and ∞ otherwise. We denote by $Occ_k^H(P,T)$ the set of k-mismatch occurrences of P in T. For brevity, we define $\mathsf{hd}(P,T^*) = \mathsf{hd}(P,T^\infty[1..|P|])$.

▶ **Fact 3.1** ([9, Lemmas 6.2-6.4]). *Consider positive integers k, n, σ such that $k \leq n$ and $\sigma = n^{O(1)}$, and a family $\mathcal{U} \subseteq \{0,\ldots,\sigma-1\}^{\leq n}$ of strings. One can assign $O(k \log n)$-bit sketches $\mathsf{sk}_k(U)$ to all strings $U \in \mathcal{U}$ so that each of the following holds, assuming that all processed strings belong to $\mathcal{U}$:*

1. *Given sketches $\mathsf{sk}_k(U), \mathsf{sk}_k(V)$, there is an algorithm that uses $O(k \log^3 n)$ time to decide whether $\mathsf{hd}(U,V) \leq k$. If so, $\mathsf{MI}(U,V)$ is reported.*
2. *There is an algorithm that constructs one of $\mathsf{sk}_k(U), \mathsf{sk}_k(V)$ or $\mathsf{sk}_k(UV)$ given the two other sketches in $O(k \log n)$ time.*
3. *There is an algorithm that constructs $\mathsf{sk}_k(U)$ or $\mathsf{sk}_k(U^m)$ given the other sketch and the integer m in $O(k \log n)$ time.*
4. *If $\mathsf{hd}(U,V) = O(k)$, there is an algorithm that constructs $\mathsf{sk}_k(V)$ from $\mathsf{sk}_k(U)$ and $\mathsf{MI}(U,V)$ in $O(k \log^2 n)$ time.*

All algorithms use $O(k \log n)$ space and succeed w.h.p.

Below, we refer to the sketch of Fact 3.1 as the *k-mismatch sketch*. Note that for a string U and all integer $k' \geq k$, the sketch $\mathsf{sk}_{k'}(U)$ includes the sketch $\mathsf{sk}_k(U)$ (see [9] for a definition). Beyond the properties above, the k-mismatch sketch has one additional property:

▶ **Corollary 3.2** ([9, Fact 4.4]). *There exists a streaming algorithm that processes a string U in $O(k \log n)$ space using $O(\log^2 n)$ time per character so that the sketch $\mathsf{sk}_k(U)$ can be retrieved on demand in $O(k \log^2 n)$ time.*

By analysing the details of [9, Corollary 3.4], one can derive a streaming algorithm for computing all occurrences of a pattern in a text when the pattern is a prefix of some string and the text is a substring of the same string, which we refer to as the *k-mismatch algorithm*:

▶ **Corollary 3.3.** *Given a string T of length n, there is a streaming algorithm for a pattern $P = T[1..\ell]$ and a text $T[i..j]$, where $1 \leq i, j, \ell \leq n$, which uses $O(k \log^2 n)$ space and takes $O(k \log^4 n)$ time per arriving character. The algorithm reports all positions p such that $i+\ell-1 \leq p \leq j$ and $\mathsf{hd}(T[p-\ell+1..p], P) \leq k$ at the moment of their arrival. For each reported position p, $\mathsf{MI}(T[p-\ell+1..p], P)$ and $\mathsf{sk}_k(T[1..p-\ell])$ can be reported on demand in $O(k \log^2 n)$ time at the moment when p arrives. The algorithm is correct w.h.p.*

Proof. Clifford et al. [9, Corollary 3.4] presented an algorithm that reports the endpoints of all k-mismatch occurrences of a pattern in a text assuming that it first receives the pattern as a stream and then a text as a stream as well. The algorithm uses $O(k \log^2 n)$ space and takes $O(k \log^4 n)$ time per arriving character and is correct w.h.p.

This is not the current best algorithm (presented in Clifford et al. [9] as well). The reason why we selected the simpler algorithm is that it allows for the necessary preprocessing of the pattern to be easily done before one needs it for the text processing. Namely, the only information the algorithm stores about the pattern are the k-mismatch sketches of the prefixes of the pattern of the lengths equal to powers of two, and the k-mismatch sketch of the pattern itself, computed via Corollary 3.2, and the k-mismatch sketch of the pattern's prefix of length ℓ is never used before the position $\ell+1$ of the text. ◀

▶ **Fact 3.4** (cf. [8, Theorems 3.1 and 3.2]). *Given a pattern P of length m, a text T of length $n \leq \frac{3}{2}m$, and a threshold $k \in \{1, \ldots, m\}$, at least one of the following holds:*
1. *The number of k-mismatch occurrences of P in T is bounded by $576 \cdot n/m \cdot k$.*
2. *There is a primitive string Q of length $|Q| \leq m/128k$ that satisfies $\mathsf{hd}(P, Q^*) \leq 2k$.*

In the second case, the difference between the starting positions of any two k-mismatch occurrences of P in T is a multiple of $|Q|$ and if T' is the minimal substring of the text containing all k-mismatch occurrences of P in T, then $\mathsf{hd}(T', Q^*) \leq 6k$.

Let S be a string of length n. For our next lemma, we introduce the *forward cyclic rotation* $\mathsf{rot}(S) = S[n]S[1] \ldots S[n-1]$. In general, for $s \in \mathbb{N}$, a cyclic rotation $\mathsf{rot}^s(S)$ with shift s (resp. $-s$) is obtained by iterating rot (resp. rot^{-1}) s times. Note that a string S is primitive if and only if $\mathsf{rot}^s(S) = S$ implies $s = 0 \pmod{|S|}$.

▶ **Lemma 3.5.** *Given two strings X, Y such that X is a prefix of Y and $|Y| \leq \frac{5}{2}|X|$. Assume that there are primitive strings Q_X such that $\mathsf{hd}(Q_X^*, X) \leq k$ and $|Q_X| \leq \frac{|X|}{128k}$ and Q_Y such that $\mathsf{hd}(Q_Y^*, Y) \leq k$ and $|Q_Y| \leq \frac{|Y|}{128k}$. We then have $Q_X = Q_Y$.*

Proof. Suppose towards a contradiction that $Q_X \neq Q_Y$. By the triangle inequality, we have $\mathsf{hd}(Q_X^\infty[1..|X|], Q_Y^\infty[1..|X|]) \leq 2k$ and $\max\{|Q_X|, |Q_Y|\} \leq \frac{|Y|}{128k} \leq \frac{5|Y|}{256}$. Assume w.l.o.g. $|Q_X| \leq |Q_Y|$.

If there is $i \in \mathbb{N}$ such that $|Q_Y| = i|Q_X|$, then by primitivity of Q_Y, we have $\mathsf{hd}(Q_Y, Q_X^i) \geq 1$, and $\mathsf{hd}(Q_Y^\infty[1..|X|], Q_X^\infty[1..|X|]) \geq \frac{|X|}{|Q_Y|} \geq \frac{128k|X|}{|Y|} \geq \frac{256}{5}k > 2k$. Otherwise, for all $1 \leq j \leq |Q_X|$ we have

$$\begin{aligned}
&\mathsf{hd}(Q_Y^2, \mathsf{rot}^j(Q_X)^*) = \\
&= \mathsf{hd}(Q_Y, \mathsf{rot}^j(Q_X)^\infty[1..|Q_Y|]) + \mathsf{hd}(Q_Y, \mathsf{rot}^j(Q_X)^\infty[|Q_Y|+1..2|Q_Y|]) = \\
&= \mathsf{hd}(Q_Y, \mathsf{rot}^j(Q_X)^\infty[1..|Q_Y|)) + \mathsf{hd}(Q_Y, \mathsf{rot}^{j+|Q_Y|}(Q_X)^\infty[1..|Q_Y|]) \geq 1
\end{aligned}$$

The last inequality holds because $j \neq j + |Q_Y| \pmod{|Q_X|}$ and Q_X is primitive. Hence, $\mathsf{hd}(Q_Y^\infty[1..|X|], Q_X^\infty[1..|X|]) \geq \frac{|X|}{2|Q_Y|} \geq \frac{128}{5}k > 2k$, and $Q_Y = Q_X$. ◀

3.2 Structure of the algorithm

We can test a position in the following way to decide whether it is a k-mismatch period:

▶ **Proposition 3.6.** *Given $\mathsf{sk}_k(T)$, $\mathsf{sk}_k(T[1..p-1])$ and $\mathsf{sk}_k(T[1..n-p+1])$ for an integer $1 \leq p \leq n$, there is an algorithm that can decide whether p is a k-mismatch period of T using $O(k \log^3 n)$ time and $O(k \log n)$ space. In this case, it also returns $\mathsf{MI}(T[p..], T[..n-p+1])$. The algorithm is correct w.h.p.*

Proof. First, the algorithm applies Fact 3.1 to compute $\mathsf{sk}_k(T[p..])$ from $\mathsf{sk}_k(T)$ and $\mathsf{sk}_k(T[1..p-1])$ in $O(k \log n)$ time and space. By Observation 3.7, p is a k-mismatch period of T iff $h = \mathsf{hd}(T[1..n-p+1], T[p..n]) \leq k$. Given $\mathsf{sk}_k(T[p..])$ and $\mathsf{sk}_k(T[1..n-p+1])$, Fact 3.1 allows to decide whether $h \leq k$ in $O(k \log^3 n)$ time and $O(k \log n)$ space. ◀

For $p \in [n/2+1..n/2+\Delta]$, our algorithm computes the sketches required by Proposition 3.6 via Fact 3.1 and the weights $w(T[p..])$ via Proposition 2.9 using $O(nk \log n + \Delta t_w(n))$ total time and $O(\Delta(k \log n + s_w(n)))$ space. After reaching the end of T, the algorithm tests each of the candidates $p \in [n/2+1..n/2+\Delta]$ in $O(\Delta k \log^3 n) = O(nk \log^3 n)$ total time and $O(k \log n)$ space, and for each k-mismatch period, returns p, $w(T[p..])$, and $\mathsf{MI}(T[p..], T[..n-p+1])$.

The rest of the section is devoted to computing periods $p \in [1..n/2]$. The following simple observation is crucial for the correctness of our algorithm.

▶ **Observation 3.7.** *An integer $1 \le p \le n$ is a k-mismatch period of T iff* $\mathsf{hd}(T[1..n-p+1], T[p..n]) \le k$. *As a corollary, if p is a k-mismatch period of T, then for all $1 \le \ell \le n-p$, the position p is the starting position of a k-mismatch occurrence of $T[1..\ell+1]$.*

It follows that we can use k-mismatch occurrences of appropriately chosen prefixes of T in T to filter out candidate k-mismatch periods. For $j = 1, \ldots, \lceil \log_{3/2} n \rceil$, define $\ell_j := \lfloor n/(3/2)^j \rfloor$, $P_j = T[1..\ell_j]$, and $T_j = T[\max\{\lfloor n/2 \rfloor - \ell_j, 1\}..\lfloor n/2 \rfloor - \ell_{j+1} + \ell_j - 2]$. For each j, we give two algorithms run in parallel, which together compute the set $\mathcal{P}_k^j$ of all k-mismatch periods of T in the interval $[\lfloor n/2 \rfloor - \ell_j..\lfloor n/2 \rfloor - \ell_{j+1} - 1]$. The first algorithm assumes that the number of k-mismatch occurrences of P_j in T_j is at most $K = 576k$ (we call such P_j "non-periodic", slightly abusing the standard definition), while the second one is correct when the number of occurrences is larger than K (we call such P_j "periodic").

3.3 The algorithm for non-periodic P_j

We maintain the sketch and the weight of the current text T using Corollary 3.2 and Proposition 2.9 in $O(\log^2 n + t_w(n))$ time per character and $O(k \log n + s_w(n))$ space. After reading $P_j = T[1..\ell_j]$, we memorise $w(P_j)$. Additionally, we run the k-mismatch algorithm (Corollary 3.3) for a pattern P_j and a text T_j, which in particular computes $\mathsf{sk}_k(P_j)$. Furthermore, we maintain two hash tables, each of size at most K, Pref_j and Suf_j. Intuitively, we want Pref_j to contain every position p which is the starting position of a k-mismatch occurrence of P_j in T_j, associated with $\mathsf{sk}_k(T[1..p-1])$ and the weight of $T[1..p-1]$. As for the table Suf_j, we would like it to contain every position t such that $n-t+1 \in \mathrm{Pref}_j$, again associated with $\mathsf{sk}_k(T[1..t])$ and the weight of $T[1..t]$. We implement Pref_j and Suf_j via the cuckoo hashing scheme [22] and de-amortise as explained in [2, Theorem A.1] to yield the following:

▶ **Fact 3.8** ([22, 2])**.** *A set of K integers in $\{0,1\}^w$, where $w = \Theta(\log n)$ is the size of the machine word, can be stored in $O(K \log n)$ space while maintaining look-up queries in $O(1)$ worst-case time and insertions in $O(1)$ worst-case time w.h.p.*

When we receive a character $T[p]$, the tables are updated as follows. Assume first that the k-mismatch algorithm detects a new occurrence of P_j ending at the position p. We retrieve $\mathsf{sk}_k(T[1..p-\ell_j])$ and $\mathsf{MI}(P_j, T[p-\ell_j+1..p])$ in $O(k \log^2 n)$ time (Fact 3.1). Furthermore, with $w(P_j)$, we can deduce $w(T[p-\ell_j+1..p])$, and finally, with $w(T[..p])$, we compute $w(T[1..p-\ell_j])$, and add $p-\ell_j$ associated with $\mathsf{sk}_k(T[1..p-\ell_j])$ and $w(T[1..p-\ell_j])$ to Pref_j in $O(kt_w(n))$ time and $O(s_w(n))$ space. Secondly, if for $t = n-p$ we have $t \in \mathrm{Pref}_j$, we add p associated with $\mathsf{sk}_k(T[1..p])$ to Suf_j. If either of the two insertions takes more than constant time or if the size of any of Pref_j and Suf_j becomes larger than K, the algorithm terminates and returns $\perp$. Assume that the algorithm has reached the end of T.

▶ **Proposition 3.9.** *If $t \in \mathcal{P}_k^j$, then $t-1 \in \mathrm{Pref}_j$ and $n-t+1 \in \mathrm{Suf}_j$.*

Proof. As $t \in \mathcal{P}_k^j$, by Observation 3.7 $t + \ell_j - 1$ is the ending position of a k-mismatch occurrence of P_j in T. Furthermore, $t \in [\lfloor n/2 \rfloor - \ell_j..\lfloor n/2 \rfloor - \ell_{j+1} - 1]$, and hence $T[t, t+\ell_j-1]$ is a fragment of T_j. This proves that $t-1 \in \mathrm{Pref}_j$. To show that $n-t+1 \in \mathrm{Suf}_j$, note that

$$2t \le 2(\lfloor n/2 \rfloor - \ell_{j+1} - 1) \le n - 2\ell_{j+1} - 2 \le n - \ell_j.$$

Therefore, $t + \ell_j - 1 < n - t + 1$, and $t-1$ will be added to Pref_j before $n-t+1$. The claim follows. ◀

Finally, the algorithm considers each position $t \in \mathrm{Pref}_j$, extracts $\mathsf{sk}_k(T[1..t])$ from Pref_j and $\mathsf{sk}_k(T[1..n-t])$ from Suf_j and if t passes the test of Proposition 3.6, reports $t+1$ as a k-mismatch period of T, and returns $w(T[t+1..]) = w(T) - w(T[1..t])$.

▶ **Proposition 3.10.** *Assume that the number of occurrences of P_j in T_j is at most $K = 576k$. The algorithm computes $\mathcal{P}_k^j$ and $\mathsf{MI}(T[p..], T[..n-p+1]), p \in \mathcal{P}_k^j$ in $O(n(k\log^4 n + t_w(n)))$ time and $O(k^2\log^2 n + s_w(n))$ space and is correct w.h.p.*

Proof. The algorithm runs an instance of the k-mismatch algorithm (Corollary 3.3) that takes $O(nk\log^4 n)$ time and $O(k\log^2 n)$ space. Computing weights takes $O(nt_w(n))$ time and $s_w(n)$ space. Adding elements to Pref_j and Suf_j, as well as look-ups, takes $O(1)$ time per element, and we add at most $K = O(k)$ elements in total. The two hash tables occupy $O(k^2\log^2 n)$ space (Fact 3.8, Fact 3.1). Finally, testing all candidate positions requires $O(K \cdot k\log^3 n) = O(nk\log^3 n)$ time and $O(k\log n)$ space (Proposition 3.6). The correctness of the algorithm follows from Proposition 3.9 and Proposition 3.6. The algorithm can fail if the k-mismatch algorithm errs, or if adding an element to the hash tables takes more than constant time, or if the test fails. By the union bound, Corollary 3.3, Fact 3.1, and Proposition 3.6, the failure probability is inverse-polynomial in n. ◀

3.4 The algorithm for periodic P_j

We first explain how we preprocess P_j. Recall the Boyer–Moore majority vote algorithm:

▶ **Fact 3.11** ([7])**.** *Given a sequence $e_1, \ldots, e_m$ of elements, there is a streaming algorithm that stores $O(1)$ elements of the sequence and returns a majority element if there exists one (otherwise, it can return an arbitrary element). Assuming constant-time access and comparison on the elements, the algorithm takes $O(m)$ time.*

▶ **Lemma 3.12.** *Given a prefix $P = T[1..\ell]$ of T, there is a streaming algorithm that uses $O(k(s_w(n) + \log^2 n))$ space and runs in $O(\ell k\log^4 n + \ell \cdot t_w(n) + k^2\log^2 n)$ time. If there is a primitive string Q of length $|Q| \le \frac{\ell}{128k}$ that satisfies $\mathsf{hd}(P, Q^*) < 2k$, the algorithm computes, correctly w.h.p., $|Q|$, $\mathsf{sk}_{3k}(Q)$, and $w(Q)$ before or upon the arrival of $T[(\lfloor \ell/|Q| \rfloor - 2) \cdot |Q|]$. If there is no such string, the algorithm determines it before $T[\ell]$ arrives.*

Proof. The main idea of the lemma is that $|Q|$ must be the starting position of the first $\Theta(k)$-mismatch occurrence of $P[1..\lfloor \frac{\ell}{2} \rfloor]$ in $T[2..]$. As soon as we know $|Q|$, we compute the $3k$-mismatch sketches and the weights of $\Theta(k)$ consecutive substrings of length $|Q|$. By Fact 3.4, the majority of them equal Q, which allows computing $\mathsf{sk}_{3k}(Q)$ and $w(Q)$ via the Boyer–Moore majority vote algorithm [7]. We now provide full details.

For brevity, let $\ell' = \lfloor \frac{\ell}{2} \rfloor$. We run the $8k$-mismatch algorithm for a pattern $P[1..\ell']$ and a text $T[2..]$. If the $8k$-mismatch algorithm does not detect an occurrence of P before or when reading the position $\ell' + \frac{\ell}{128k} < \ell$, the algorithm concludes that Q does not exist and terminates. Assume now that the algorithm does detect a $8k$-mismatch occurrence of the pattern ending at a position p, $p \le \ell' + \frac{\ell}{128k}$, and let it be the first detected occurrence. The instance of the $8k$-mismatch algorithm is immediately terminated and we launch the majority vote algorithm (Fact 3.11). Let $q = p - \ell' + 1$ and p' the smallest multiple of q greater than p. We then compute $\mathsf{sk}_{3k}(T[p'+1..p'+q])$, $\mathsf{sk}_{3k}(T[p'+q+1..p'+2q])$, $\ldots$, $\mathsf{sk}_{3k}(T[p'+(12k-1)q+1..p'+12kq])$ via Corollary 3.2 and feed them into the majority vote algorithm. After all the $12k$ sketches have been computed, which happens when we read the position $p' + 12kq \le (\lfloor \ell/q \rfloor - 2) \cdot q$, we return q and the output of the majority vote algorithm as $\mathsf{sk}_{3k}(Q)$. Using the same majority vote approach, we compute $w(Q)$.

We now show the correctness of the algorithm. We start by showing that if Q exists, then $q = |Q|$. Let i be a multiple of $|Q|$ smaller than $\ell/2$. By Fact 3.4 and the triangle inequality, we have $\mathsf{hd}(P[1..\ell'], T[i+1..i+\ell']) \leq \mathsf{hd}(P[1..\ell'], Q^*) + \mathsf{hd}(Q^*, T[i+1..i+\ell']) \leq 8k$. Reciprocally, if i is not a multiple of $|Q|$, then by primitivity of Q we have $\mathsf{hd}(Q^\infty[1..\ell'], Q^\infty[i+1..i+\ell']) \geq \lfloor \ell'/|Q| \rfloor \geq 62k$. Furthermore, by the triangle inequality, $\mathsf{hd}(Q^\infty[1..\ell'], Q^\infty[i+1..i+\ell']) \leq 8k + \mathsf{hd}(P[1..\ell'], T[i+1..i+\ell'])$, which implies $\mathsf{hd}(P[1..\ell'], T[i+1..i+\ell']) \geq 54k$. Consequently, if Q exists, at least $6k$ of the strings $T[p'+1..p'+q]$, $T[p'+q+1..p'+2q]$, $\ldots$, $T[p'+(12k-1)q+1..p'+12kq]$ are equal to Q, and the majority vote algorithm indeed outputs $sk_{3k}(Q)$ (assuming that neither the $8k$-mismatch algorithm nor the algorithm of Corollary 3.2 did not err, which is true w.h.p.).

We finally analyse the complexity of the preprocessing step. The $8k$-mismatch pattern matching algorithm (Corollary 3.3) takes $O(k \log^4 n)$ time per character and $O(k \log^2 n)$ space. The algorithm of Corollary 3.2 uses $O(\log^2 n)$ time per character $O(k \log n)$ space. Furthermore, $O(k)$ sketches are retrieved, which takes $O(k^2 \log^2 n)$ time (Corollary 3.2) and maintaining weights requires $t_w(n)$ time and $k \cdot s_w(n)$ space. Finally, the majority vote algorithm takes $O(k \log n)$ space and $O(k \log n)$ time. In total, the algorithm takes $O(k(s_w(n) + \log^2 n))$ space and $O(\ell k \log^4 n + \ell \cdot t_w(n) + k^2 \log^2 n)$ time. ◀

We now apply the lemma above to preprocess P_j as follows. We maintain the $3k$-mismatch sketch of T using Corollary 3.2. Fact 3.4 ensures that if there are at least K occurrences of P_j in T_j, then there exists a primitive string Q of length $q := |Q| \leq \frac{\ell_j}{128k}$ such that $\mathsf{hd}(Q^*, P_j) < 2k$. For brevity, let $\lambda_j = (\lfloor \ell_j/q \rfloor - 2) \cdot q$. We apply Lemma 3.12 to P_j and condition on the fact that it outputs q, $\mathsf{sk}_{3k}(Q)$, and $w(Q)$ before or upon the arrival of $T[\lambda_j]$. By an application of Fact 3.1, we compute $\mathsf{sk}_{3k}(Q^{\lfloor \lambda_j/q \rfloor + 1})$ and then $\mathsf{MI}(P_j[..\lambda_j+q], Q^{\lfloor \ell_j/q \rfloor + 1})$ using $\mathsf{sk}_{3k}(P_j[..\lambda_j+q])$. To finish the preprocessing, we compute one more sketch:

▶ **Lemma 3.13.** *Assume there are $\geq K$ k-mismatch occurrences of P_j in T_j. There is a streaming algorithm that uses $O(k \log^2 n + s_w(n))$ space and $O(k \log^4 n + t_w(n))$ time per character, and computes $\mathsf{sk}_k(Q[..r])$ for $r := n - 2p \pmod q$ and $w(Q[..r])$ correctly w.h.p. upon arrival of $T[p]$, where p is the endpoint of the first k-mismatch occurrence of P_j in T_j.*

Proof. By the condition of the lemma, Q is defined and we assume to have computed it by arrival of $T[\lambda_j]$, where $\lambda_j = (\lfloor \ell_j/q \rfloor - 2) \cdot q$ and $q = |Q|$. We run two instances of the k-mismatch algorithm: One for a pattern $P_j[..\lambda_j]$ and T_j, and the other for P_j and T_j. Assume that we detect a k-mismatch occurrence of $P_j[..\lambda_j]$ ending at a position x. Let $r' = n - 2(x + \ell_j - \lambda_j) \pmod q$. We compute $\mathsf{sk}_k(T[x+1..x+r'])$ via Corollary 3.2, and $w(T[x+1..x+r'])$ via Proposition 2.9. If $p = x + \ell_j - \lambda_j$ is not the ending position of a k-mismatch occurrence of P_j, we discard the computed information and continue. Otherwise, we have $r' = r$. At the position p the k-mismatch algorithm extracts $\mathsf{MI}(T[p - \ell_j + 1..p], P_j)$ in $O(k)$ time, and we use it to extract $\mathsf{MI}(T[x+1..x+r], Q[..r])$ from $\mathsf{MI}(P_j[..\lambda_j + q], Q^*)$ in $O(k)$ time as well. Note that the size of the extracted mismatch information is at most $6k$ by Fact 3.4. Finally, we apply Fact 3.1 to compute $\mathsf{sk}_k(Q[..r])$ from $\mathsf{sk}_k(T[x+1..x+r])$ and $\mathsf{MI}(T[x+1..x+r], Q[..r])$ in $O(k \log^2 n)$ time and $O(k \log n)$ space. Similarly, with $w(T[x+1..x+r])$ and the mismatch information, we compute $w(Q[..r])$ in $O(kt_w(n))$ time and $O(s_w(n))$ space. (See Figure 2 for an illustration.)

To show the complexity of the algorithm, we need to understand the structure of the k-mismatch occurrences of $P_j[..\lambda_j]$ in T_j. As each k-mismatch occurrence of P_j starts with a k-mismatch occurrence of $P_j[..\lambda_j]$, T_j contains at least K k-mismatch occurrences of the latter. Since $|T_j| \leq 2\ell_j - \ell_{j-1} \leq \frac{3}{2}\lambda_j$, by Fact 3.4 there is a primitive string Q' such that $|Q'| \leq \frac{\lambda_j}{128k}$ and $\mathsf{hd}(P_j[..\lambda_j], (Q')^*) \leq 2k$. By Lemma 3.5, $Q = Q'$, and again by Fact 3.4,

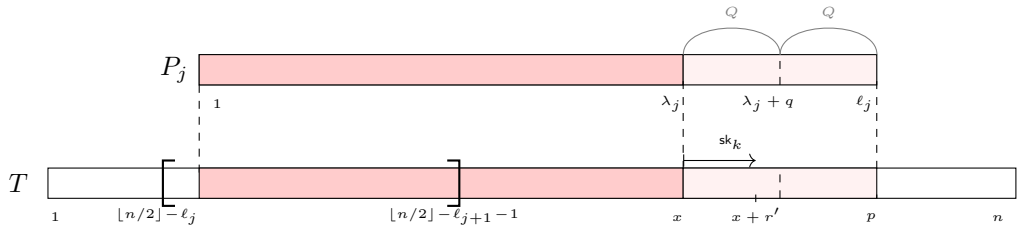

Figure 2 When we detect a k-mismatch occurrence of $P_j[..\lambda_j]$, we use the next q characters to compute the candidate for $\mathsf{sk}_k(Q[..r])$. If the k-mismatch occurrence of $P_j[..\lambda_j]$ extends to a k-mismatch occurrence of P_j, then we keep the candidate sketch.

the difference between the starting positions of any two k-mismatch occurrences of $P_j[1..\lambda_j]$ in T_j is a multiple of q. Therefore, we never process more than two k-mismatch occurrences of $P_j[..\lambda_j]$ at a time and the bounds follow. ◀

This information will allow us computing the sketches necessary for Proposition 3.6.

3.4.1 Main phase of the algorithm

The main phase distinguishes two cases: $j = 1, 2$ and $j \geq 3$. For $j = 1, 2$, we show the following result:

▶ **Proposition 3.14.** *Assume that $j = \{1, 2\}$ and that P_j has more than $K = 576k$ k-mismatch occurrences in T_j. The algorithm computes $\mathcal{P}_k^j$, $\mathsf{MI}(T[p..], T[..n - p + 1])$ and $w(T[p..])$ for $p \in \mathcal{P}_k^j$ in $O(n \cdot kt_w(n) \log^4 n)$ time and $O(k^2 \log^2 n + s_w(n))$ space and is correct w.h.p.*

Proposition 3.14 can be proven using the same ideas as in the case $j \geq 3$, but needs some additional care because the size of the pattern being large ($n/4$ or $n/2$) leads to edge cases that are treated separately. Below, we focus on the case $j \geq 3$. We start by extending P_j into a prefix P'_j (Algorithm 1).

Algorithm 1 Extension of P_j into P'_j of length ℓ'_j: case $j \geq 3$.

```
ℓ'_j ← λ_j, sk^1 ← sk_{3k}(T[1..λ_j]), w_1 ← w(T[1..λ_j]);
while ℓ'_j < 2ℓ_j do
    sk^2 ← sk^1, w_2 ← w_1;
    ℓ'_j ← ℓ'_j + q, sk^1 ← sk_{3k}(T[1..ℓ'_j]), w_1 ← w(T[1..ℓ'_j]);
    if h = MI(T[1..ℓ'_j], Q*) > 2k then
        return (ℓ'_j, sk^1, sk^2, w_1, w_2);          // Use sk_{3k}(T) and sk_{3k}(Q)
return (ℓ'_j, sk^1, sk^2, w_1, w_2);
```

The following inequalities are essential for analysis of correctness of the algorithm:

▶ **Proposition 3.15.** *For $j \geq 3$, we have $\ell'_j < n$ and $\lfloor n/2 \rfloor - \ell_{j+1} - 1 + (\ell'_j - 1) \leq n$.*

Proof. We start by showing the first inequality:

$$\ell'_j < 2\ell_j + q \leq \ell_j \cdot (2 + 1/128k) \leq (8n/27) \cdot (2 + 1/128k) \leq n$$

To show the second inequality, note that

$$\lfloor n/2 \rfloor - \ell_{j+1} - 1 + \ell'_j - 1 \leq$$
$$\lfloor n/2 \rfloor - n/(3/2)^{j+1} + 2n/(3/2)^j + q \leq \lfloor n/2 \rfloor + (4/3 + 1/128) \cdot n/(3/2)^j \leq n \quad \blacktriangleleft$$

We now discuss how to implement Algorithm 1. As we know $\mathsf{sk}_{3k}(Q)$ before or upon the arrival of $T[\lambda_j]$, Algorithm 1 can be implemented in streaming via Corollary 3.2, Fact 3.1 to use $O(k \log n + s_w(n))$ space and $O(k \log^2 n + t_w(n))$ time per character. Furthermore, it always terminates before the arrival of $T[n]$ (Proposition 3.15) and outputs $\ell'_j := |P'_j|$, $\mathsf{sk}_{3k}(P'_j)$ and $w(P'_j)$, and $\mathsf{sk}_{3k}(P'_j[1..\ell'_j - q])$ and $w(P'_j[1..\ell'_j - q])$. Define $T'_j = T[\max\{\lfloor n/2 \rfloor - \ell_j, 1\}..\lfloor n/2 \rfloor - \ell_{j+1} + \ell'_j - 2]$. Note that T'_j is well-defined by Proposition 3.15. Furthermore, let $P''_j = [1..\ell'_j - q]$, $\ell''_j = |P''_j|$, and $T''_j = T[\lfloor n/2 \rfloor - \ell_j..\lfloor n/2 \rfloor - \ell_{j+1} + \ell''_j - 2]$. From Observation 3.7 and Proposition 3.15 we obtain:

▶ **Corollary 3.16.** *The set $\mathcal{P}^j_k$ is a subset of the set of the starting positions of k-mismatch occurrences of P'_j in T'_j, and consequently of the set of the starting positions of k-mismatch occurrences of P''_j in T''_j.*

Consider now two subcases depending on the line where Algorithm 1 executes the return.

3.4.1.1 Return is executed in Line 6

▶ **Proposition 3.17.** *The prefixes P'_j and P''_j have the following properties:*
1. *The number of k-mismatch occurrences of P'_j in T'_j is $O(k)$.*
2. *The distance between any two k-mismatch occurrences of P''_j in T''_j is a multiple of q.*

Proof. To show the first part of the claim, note that $\ell_j \leq \ell'_j \leq \frac{5}{2}\ell_j$ and $|T'_j| \leq \ell_j - \ell_{j+1} + \ell'_j \leq \frac{3}{2}\ell'_j$. Assume, for sake of contradiction, that there are more than $576k$ occurrences of P'_j in T'_j. By Fact 3.4, there is a primitive string Q', $|Q'| \leq \ell'_j/128k$, such that $\mathsf{hd}(P'_j, (Q')^*) \leq 2k$. By Lemma 3.5, $Q' = Q$, a contradiction.

We now show the second part of the claim. Note that $\frac{2}{3}\ell_j \leq \ell_j - q \leq \ell''_j \leq 2\ell_j$ and hence $|T''_j| \leq \ell_j - \ell_{j+1} + \ell''_j \leq \frac{3}{2}\ell''_j$. Next, we have two cases: $\ell''_j \leq \ell_j$ and $\ell''_j > \ell_j$. In the first case, every position p which is a starting position of a k-mismatch occurrence of P_j in T_j is also a starting position of a k-mismatch occurrence of P''_j in T''_j. Hence, there are at least $K = 576k$ k-mismatch occurrences of P''_j in T''_j, and by Fact 3.4, there is a primitive string Q'', $|Q''| \leq \ell''_j/128k$, such that $\mathsf{hd}(P''_j, (Q'')^*) \leq 2k$. By Lemma 3.5, $Q'' = Q$. If $\ell''_j > \ell_j$, then $\mathsf{hd}(P''_j, Q^*) \leq 2k$ by construction. Consequently, by applying Fact 3.4 one more time, we obtain that the difference between the starting positions of any two k-mismatch occurrences of P''_j in T''_j is q. ◀

We apply the k-mismatch algorithm to detect k-mismatch occurrences of P''_j and P'_j in the text T'_j. In parallel, we maintain two hash tables of size at most $K = 576k$ each, Pref_j and Suf_j, implemented via Fact 3.8. When we receive a character $T[p]$, the tables are updated as follows. Assume first that the k-mismatch algorithm detects a new occurrence of P''_j ending at the position p. We retrieve $\mathsf{sk}_{3k}(T[1..p - \ell''_j])$ and $\mathsf{MI}(P''_j, T[p - \ell''_j + 1..p])$. From the mismatch information, $w(P''_j)$ and $w(T[..p])$ we compute $w(T[1..p - \ell''_j])$, and we memorise $t = p - \ell''_j$ associated with the sketch and the weight for the next q positions. Importantly, at every moment the algorithm stores at most one position-sketch pair by Proposition 3.17. By the definition of P''_j and T''_j, the k-mismatch occurrences detected by the algorithm can only start before $\lfloor n/2 \rfloor$, and consequently $2(t+1) \leq n$, which implies $t + 1 < n - t$.

Now, assume that $t+1 < n-t \leq p$. In this case, we immediately compute $\mathsf{sk}_k(T[1..n-t])$ via the following claim and memorise it for the next q positions:

▶ **Proposition 3.18.** *Assume to be given* $\mathsf{sk}_k(T[1..t])$, $\mathsf{sk}_k(Q[r..])$, *and* $\mathsf{sk}_k(Q)$. *There is an algorithm that uses* $O(k \log n)$ *space and* $O(k \log^3 n)$ *time and computes* $\mathsf{sk}_k(T[1..n-t])$. *The algorithm succeeds w.h.p.*

Proof. By Corollary 3.16 and Proposition 3.17, $(n-t) - t = iq + r$ for some integer i. Consequently, the sketch can be computed as follows. First, the algorithm computes $\mathsf{MI}(P''_j, T[p - \ell''_j + 1..p])$ and $\mathsf{MI}(P''_j, Q^*)$. Secondly, it computes $\mathsf{sk}_k(Q^i Q[1..r])$ and then deduces $\mathsf{sk}_k(T[t+1..n-t])$ in $O(k \log n)$ space and $O(k \log^3 n)$ time via Fact 3.1. Finally, it computes $\mathsf{sk}_k(T[1..n-t])$ from $\mathsf{sk}_k(T[1..t])$ and $\mathsf{sk}_k(T[t+1..n-t])$. ◀

Also, if the current position p equals $n-t$, where t is the position in the stored position-sketch pair, we memorise $\mathsf{sk}_k(T[1..n-t])$. Again, by Proposition 3.17 the algorithm stores at most one sketch at a time. If the current position p is the endpoint of a k-mismatch occurrence of P'_j in T'_j, the position $p-q$ is necessarily the endpoint of a k-mismatch occurrence of P''_j in T''_j, and we store $t = p - q - \ell''_j$ associated with $\mathsf{sk}_k(T[1..t])$ and $w(T[1..t])$. We add this triple to Pref_j. In addition, if $n - t \leq p$, we have already computed $\mathsf{sk}_k(T[1..n-t])$, and we add it to Suf_j. Finally, if p is the current position and for $t = n - p$ we have $(t, \mathsf{sk}_k(T[1..t]), w(T[1..t])) \in \mathrm{Pref}_j$, we add p associated with $\mathsf{sk}_k(T[1..p])$ to Suf_j.

If any of the insertions takes more than constant time or if the size of any of Pref_j and Suf_j becomes larger than K, the algorithm terminates and returns $\perp$.

When the entire string T has arrived, the algorithm considers each position $t \in \mathrm{Pref}_j$, extracts $\mathsf{sk}_k(T[1..t]), w(T[1..t])$ from Pref_j and $\mathsf{sk}_k(T[1..n-t])$ from Suf_j and if t passes the test of Proposition 3.6, reports $t+1$ as a k-mismatch period of T, and also returns $w(T[t+1..]) = w(T) - w(T[1..t])$ (undefined if one of the values on the right is undefined), and $\mathsf{MI}(T[t+1..], T[..n-t])$.

3.4.1.2 Return is executed in Line 7

▶ **Proposition 3.19.** *If return is executed in Line 7, we have* $\mathsf{hd}(P'_j, Q^*) < 2k$ *and for all* $t \in [\lfloor n/2 \rfloor - \ell_j .. \lfloor n/2 \rfloor - \ell_{j+1} - 1]$ *there is* $\lfloor n/2 \rfloor \leq n - t \leq \lfloor n/2 \rfloor - \ell_j + (\ell'_j + 2)$.

Proof. The first part of the claim is immediate by construction. To show the second part, recall that $\ell'_j \geq 2\ell_j$. Hence,

$$\lfloor n/2 \rfloor \leq n - t \leq n - \lfloor n/2 \rfloor + \ell_j + 1 \leq \lfloor n/2 \rfloor - \ell_j + (2\ell_j + 2) \leq \lfloor n/2 \rfloor - \ell_j + (\ell'_j + 2)$$ ◀

We run the k-mismatch algorithm (Corollary 3.3) for P'_j and T'_j. If a position p is the endpoint of the first k-mismatch occurrence of P'_j in T'_j, we retrieve $\mathsf{sk}_k(T[1..p - \ell'_j])$ and $\mathsf{MI}(T[p - \ell'_j + 1..p], P'_j)$, and deduce $w(T[1..p - \ell'_j])$ with the same method as in the previous case. Also, we memorise p, the sketch, the mismatch information and the weight. We would now like to process the last k-mismatch occurrence of P'_j in T'_j in a similar way. As it is not possible to say in advance whether the current k-mismatch occurrence is the last one, we instead do the following. Starting from the second endpoint p' of a k-mismatch occurrence of P'_j in T'_j, we retrieve $\mathsf{MI}(T[p' - \ell'_j + 1..p'], P'_j)$ by Corollary 3.3, and memorise p' and the mismatch information until the next k-mismatch occurrence is detected, when they are discarded. Additionally, at the position $p'+1$ we launch a new instance of the algorithm of Corollary 3.2, maintaining $\mathsf{sk}_k(T[p'+1..])$. Again, if we detect another k-mismatch occurrence, we discard the currently stored sketch. In addition, when $T[x]$ arrives, for

$x \in \{\lfloor n/2 \rfloor - \ell_j + (\ell'_j + 1), \lfloor n/2 \rfloor - \ell_j + (\ell'_j + 2)\}$ we launch a new instance of the algorithm in Corollary 3.2, maintaining $\mathsf{sk}_k(T[x+1..])$. This way, when we reach the end of T, we have the following information at hand:

1. For the endpoint p of the first k-mismatch occurrence of P'_j in T_j, $\mathsf{sk}_k(T[1..p-\ell'_j])$, weight $w(T[1..p-\ell'_j])$, and the mismatch information $\mathsf{MI}(T[p-\ell'_j+1..p], P'_j)$;
2. For the endpoint p' of the last k-mismatch occurrence of P'_j in T_j, the mismatch information $\mathsf{MI}(T[p'-\ell'_j+1..p'], P'_j)$ and $\mathsf{sk}_k(T[p'+1..])$;
3. $\mathsf{sk}_k(T[x+1..])$ for $x \in \{\lfloor n/2 \rfloor - \ell_j + (2\ell_j + 1), \lfloor n/2 \rfloor - \ell_j + (2\ell_j + 2)\}$.

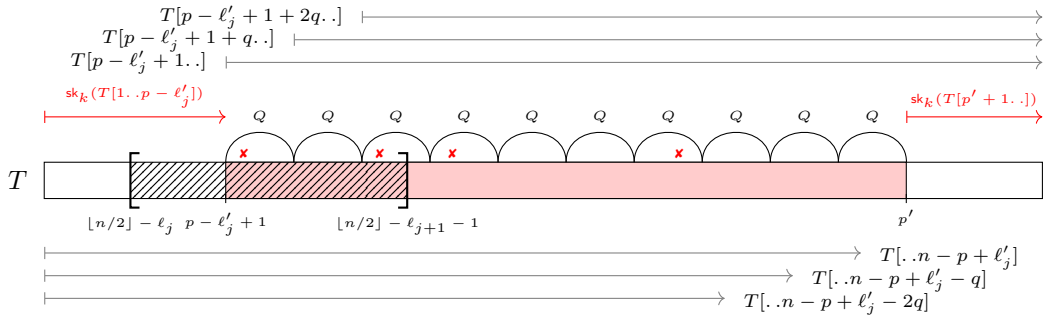

Figure 3 If Algorithm 1 executes return in Line 7, $\mathcal{P}^k_j \subseteq \{p-\ell'_j+1, p+q-\ell'_j+1, \ldots, p'-\ell'_j+1\}$. To test each position in the latter set, we exploit the structure of $T[p-\ell'_j+1..p'-\ell'_j+1]$ (shown in red, with crosses marking mismatches between it and Q^∞).

By Proposition 3.19 and Fact 3.4, we have $\mathcal{P}^k_j \subseteq \{p-\ell'_j+1, p+q-\ell'_j+1, \ldots, p'-\ell'_j+1\}$. We test each position t in the latter set as follows (see Figure 3):

1. **Compute $\mathsf{sk}_k(T[t..])$.** First, retrieve $\mathsf{sk}_k(T[1..t-1])$ from sketches $\mathsf{sk}_k(T[1..p-\ell'_j])$ and $\mathsf{sk}_k(Q^{(t-p)/q})$, and the mismatch information $\mathsf{MI}(T[p-\ell'_j+1..p], P'_j)$, $\mathsf{MI}(T[p'-\ell'_j+1..p'], P'_j)$, and $\mathsf{MI}(P'_j, Q^*)$ in $O(k \log^2 n)$ time and $O(k \log n)$ space via Fact 3.1. Second, compute $\mathsf{sk}_k(T[t..])$ from $\mathsf{sk}_k(T)$ and $\mathsf{sk}_k(T[1..t-1])$ in $O(k \log^2 n)$ time and $O(k \log n)$ space via another application of Fact 3.1.
2. **Compute $w(T[t..])$.** First, retrieve $w(T[1..t-1])$ from $w(T[1..p-\ell'_j])$ and $w(Q^{(t-p)/q}) = [(t-p)/q] \cdot w(Q)$, and the mismatch information $\mathsf{MI}(T[p-\ell'_j+1..p], P'_j)$, $\mathsf{MI}(T[p'-\ell'_j+1..p'], P'_j)$, and $\mathsf{MI}(P'_j, Q^*)$ in $O(kt_w(n))$ time and $s_w(n)$ space. Second, compute $w(T[t..]) = w(T) - w(T[1..t-1])$ (undefined if one of the values on the right is undefined).
3. **Compute $\mathsf{sk}_k(T[1..n-t+1])$.** If $n-t+1 \in \{\lfloor n/2 \rfloor - \ell_j + (\ell_j + 1), \lfloor n/2 \rfloor - \ell_j + (\ell_j + 2)\}$, we already know the sketch. Otherwise, by Proposition 3.19 $\lfloor n/2 \rfloor \le n-t+1 \le \lfloor n/2 \rfloor + \ell'_j$. Additionally, $(n-t+1) - (p-\ell'_j+1) + 1 = q \cdot i + r$ for an integer r defined as in Lemma 3.13. Hence, $\mathsf{sk}_k(T[1..n-t+1])$ can be computed via Fact 3.1: start by computing $\mathsf{sk}_k(T[p-\ell'_j+1..n-t+1])$ from $\mathsf{sk}_k(Q)$, $\mathsf{sk}_k(Q[1..r])$, and the mismatch information for p and p', and then use $\mathsf{sk}_k(T[1..p-\ell'_j])$ to compute $\mathsf{sk}_k(T[1..n-t])$.
4. **Compute $\mathsf{hd}_{\le k}(T[t..n], T[1..n-t+1])$** using the computed sketches via Fact 3.1. If it is at most k, output t as a k-mismatch period, and return $w(T[t..])$ and $\mathsf{MI}(T[t..], T[..n-t+1])$.

▶ **Proposition 3.20.** *Assume that $j \ge 3$ and that P_j has more than $K = 576k$ k-mismatch occurrences in T_j. The algorithm computes $\mathcal{P}^j_k$, $\mathsf{MI}(T[p..], T[..n-p+1])$ and $w(T[p..])$ for $p \in \mathcal{P}^j_k$ in $O(n \cdot kt_w(n) \log^4 n)$ time and $O(k^2 \log^2 n + s_w(n))$ space and is correct w.h.p.*

Proof. The preprocessing of P_j takes $O(n \cdot (k \log^4 n + t_w(n)))$ time and $O(k \log^2 n + s_w(n))$ space. The main phase of the algorithm starts with an extension procedure (Algorithm 1), which takes $O(n(k \log^2 n + t_w(n)))$ total time and $O(k \log n + s_w(n))$ space. If the return is

executed in Line 6 of Algorithm 1, the algorithm then runs two instances of the k-mismatch algorithm (Corollary 3.3) that take $O(nk \log^4 n)$ time and $O(k \log^2 n)$ space. Adding elements to Pref_j and Suf_j, as well as look-ups, takes $O(1)$ time per element, and we add at most $K = O(k)$ elements in total. The two hash tables occupy $O(k^2 \log^2 n)$ space (Fact 3.8, Fact 3.1). Finally, testing all candidate positions requires $O(K \cdot k \log^3 n)$ time and $O(k \log n)$ space (Proposition 3.6). If the return is executed in Line 7 of Algorithm 1, the algorithm runs an instance of the k-mismatch algorithm, which takes $O(nk \log^4 n)$ time and $O(k \log^2 n)$ space, and maintains a constant number of sketches, taking $O(n \log^2 n)$ time and $O(k \log n)$ space. The process to test the candidate k-periods uses $O(k \log n + s_w(n))$ space and $O(k(\log^3 n + t_w(n)))$ time, and it is iterated $\leq n$ times.

The algorithm can fail if the preprocessing fails, if the k-mismatch algorithm errs, or if adding an element to the hash tables takes more than constant time, or if the test fails. By the union bound, Corollary 3.3, Fact 3.1, and Proposition 3.6, the failure probability is inverse-polynomial in n. ◀

Theorem 2.10 follows from Proposition 3.10, Proposition 3.20, and Proposition 3.14.

References

1 Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya. Small-space algorithms for the online language distance problem for palindromes and squares. In *ISAAC 2023*, volume 283 of *LIPIcs*, pages 10:1–10:17, 2023. doi:10.4230/LIPICS.ISAAC.2023.10.

2 Michael A. Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. Modern hashing made simple. In *SOSA 2024*, pages 363–373, 2024. doi:10.1137/1.9781611977936.33.

3 Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome recognition in the streaming model. In *STACS 2014*, volume 25, pages 149–161, 2014. doi:10.4230/LIPIcs.STACS.2014.149.

4 Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In *STOC 2023*, pages 219–232, 2023. doi:10.1145/3564246.3585239.

5 Sudatta Bhattacharya and Michal Koucký. Streaming k-edit approximate pattern matching via string decomposition. In *ICALP 2023*, volume 261 of *LIPIcs*, pages 22:1–22:14, 2023. doi:10.4230/LIPICS.ICALP.2023.22.

6 Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching, 2023. doi:10.48550/arXiv.2302.04475.

7 Robert S. Boyer and J Strother Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–118. Kluwer Academic Publishers, 1991.

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

9 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *SODA 2019*, pages 1106–1125. SIAM, 2019. doi:10.1137/1.9781611975482.68.

10 Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. In Howard J. Karloff, editor, *SODA 1998*, pages 463–472. ACM/SIAM, 1998. URL: http://dl.acm.org/citation.cfm?id=314613.314827.

11 Mohamed G. Elfeky, Walid G. Aref, and Ahmed K. Elmagarmid. Stagger: Periodicity mining of data streams using expanding sliding windows. In *ICDM 2006*, pages 188–199, 2006. doi:10.1109/ICDM.2006.153.

12 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming periodicity with mismatches. In *APPROX/RANDOM 2017*, volume 81 of *LIPIcs*, pages 42:1–42:21, 2017. doi:10.4230/LIPICS.APPROX-RANDOM.2017.42.

13 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Periodicity in data streams with wildcards. *Theory Comput. Syst.*, 64(1):177–197, 2020. doi:10.1007/S00224-019-09950-Y.

14 Funda Ergun, Hossein Jowhari, and Mert Sağlam. Periodicity in streams. In *APPROX/RANDOM 2010*, pages 545–559, 2010.

15 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming periodicity with mismatches, 2017. arXiv:1708.04381.

16 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw 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.

17 Tomohiro I. Longest Common Extensions with Recompression. In *CPM 2017*, volume 78 of *LIPIcs*, pages 18:1–18:15, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

18 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000. doi:10.1109/18.841160.

19 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

20 Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complex. Cryptol.*, 4(2):241–299, 2012. doi:10.1515/GCC-2012-0016.

21 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.

22 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/J.JALGOR.2003.12.002.

A A gap in the previous streaming algorithm for computing k-mismatch periods

In this section, we point to the specific claim in the correctness proof of the previous streaming algorithm for computing k-mismatch periods [15] which is, in our opinion, not true.

Let S be a string length n, and $1 \leq p < q \leq \frac{n}{2}$ two k-mismatch periods of S such that $k \cdot (p+q) \leq i \leq \frac{n}{2} - k \cdot (p+q)$, where i is a position of S, and $q \geq (2k+1) \cdot \mathsf{gcd}(p,q)$. The construction of [15] introduces a grid defined over a set $\{-k, \ldots, k\}^2$. A node (a,b) of the grid represents a position $i + ap + bq$ of S. For a node representing a position j, we add edges connecting it to nodes representing $j+p$, $j+q$, $j-p$, and $j-q$ (if they exist in the grid). Finally, we say that an edge (i,j) of the gird is *bad* if $S[i] \neq S[j]$.

The proof of [15, Theorem 28], one of the key elements of the streaming algorithm of Ergün et al., relies on the fact that there are a few bad edges in the grid. One of the steps in their proof of this fact is the following claim:

▶ **Proposition A.1** ([15, Claim 20]). *The nodes of the grid correspond to distinct positions of S.*

As an immediate corollary, and since p and q are both k-mismatch periods of S, they immediately derive that there are at most $2k$ bad edges in the grid.

The authors then extend their approach to the case when S has $m \in \mathbb{N}$ k-mismatch periods $p_1 < \cdots < p_m$, where $p_m \geq (2k+1) \cdot \mathsf{gcd}(p_1, p_m)$. One can construct an m-dimensional grid in a similar way to the case $m = 2$, and it is claimed that in this grid, "the total number of bad edges is at most mk" (Page 21). However, consider the string $S = \mathtt{a}^{40}\mathtt{ba}^{60}$. All integers smaller than 50 are 2-mismatch periods of S, and in this example we have $m = 50$, $k = 2$ and $\mathsf{gcd}(1, 50) = 1$, verifying the assumption $50 \geq (2k+1) \cdot \mathsf{gcd}(1, 50)$. Let $\{-2, \ldots, 2\}^{50}$ be the grid centred around the index 41 (i.e the only character $\mathtt{b}$). Similarly to the case $m = 2$, a point $(a_1, \ldots, a_{50})$ of the grid represents the position $41 + \sum i \cdot a_i$, and an edge

between nodes representing positions i, j is bad if $S[i] \neq S[j]$. Note for any $i \leq 25$, the node $(a_1, \ldots, a_{50})$ with $a_i = 2$, $a_{2i} = -1$ and $a_j = 0$ for $j \notin \{i, 2i\}$ represents the position $41 + 2i - 2i = 41$. As a result, 41 is represented by at least distinct 25 different nodes in the grid, which contradicts Proposition A.1 for $m \neq 2$. Furthermore, these nodes are connected with each of their neighbours with a bad edge. Since each node has at least 5 neighbours, there are at least 125 distinct bad edges, contradicting the upper bound on the number of bad edges which was $mk = 100$.