Few Matches or Almost Periodicity: Faster Pattern Matching with Mismatches in Compressed Texts

Karl Bringmann* Marvin Künnemann* Philip Wellnitz*†

Abstract

A fundamental problem on strings in the realm of approximate string matching is *pattern matching with mismatches*: Given a text t, a pattern p, and a number k, determine whether some substring of t has Hamming distance at most k to p; such a substring is called a k*-match*.

As real-world texts often come in compressed form, we study the case of searching for a *small* pattern p in a text t that is *compressed* by a straight-line program. This *grammar compression* is popular in the string community, since it is mathematically elegant and unifies many practically relevant compression schemes such as the Lempel-Ziv family, dictionary methods, and others. We denote by m the length of p and by n the compressed size of t. While exact pattern matching, that is, the case $k = 0$, is known to be solvable in near-linear time $\tilde{O}(n + m)$ [Jeż TALG'15], despite considerable interest in the string community, the fastest known algorithm for pattern matching with mismatches runs in time $\tilde{O}(n\sqrt{m}\,\mathrm{poly}(k))$ [Gawrychowski, Straszak ISAAC'13], which is far from linear even for very small k.

In this paper, we obtain an algorithm for pattern matching with mismatches running in time $\tilde{O}((n + m)\,\mathrm{poly}(k))$. This is near-linear in the input size for any constant (or slightly superconstant) k. We obtain analogous running time for counting and enumerating all k-matches.

Our algorithm is based on a new structural insight for approximate pattern matching, essentially showing that either the number of k-matches is very small or both text and pattern must be almost periodic. While intuitive and simple for exact matches, such a characterization is surprising when allowing k mismatches.

Keywords: stringology, string algorithms, approx. pattern matching, grammar compression

1 Introduction

Approximate Pattern Matching Exact pattern matching is the most fundamental problem on strings: Given a text t and a pattern p, determine whether p is a substring of t. *Approximate pattern matching* generalizes this problem to finding a substring of t that is "close" to p. This is very well motivated by practical applications, e.g., whenever we want to correct human spelling mistakes or so-called read errors in DNA sequencing.

In this paper, we study *pattern matching with k mismatches*: given a number k in addition to text t and pattern p, the task is to decide whether some length-$|p|$ substring of t has Hamming distance at most k to p; such a substring is called a k*-match*. This is a very well-studied problem from the realm of approximate pattern matching. A long series of works [31, 17, 5, 11, 24] brought down the best known running time[1] to $\tilde{O}((m + k\sqrt{m}) \cdot n/m)$, where n is the text length and m is the pattern length, and there is a matching lower bound for combinatorial algorithms under the combinatorial Boolean matrix multiplication conjecture [24].

This generalizes an $\tilde{O}(n\sqrt{m})$-time algorithm for pattern matching with mismatches from the 1980s [2], which works for any k, that is, even when k is as large as $\Theta(m)$. Besides this main line of work, pattern matching with mismatches has been studied in a variety of settings, e.g., in streaming algorithms (if the text is an input stream, it was shown that $O(k\,\mathrm{polylog}\,n)$ space and $O(\sqrt{k}\,\mathrm{polylog}\,n)$ time per text symbol are sufficient [11, 13]), in dynamic data structures (with updates being either restricted to the text [4] or being allowed to occur in both the text and the pattern [12]), with "don't care" symbols [10], or with additional normalization transformations applied at every alignment [9].

Grammar Compression In this paper we consider the setting where the text t comes in compressed form. This is well motivated not only for natural language texts, which are known to be very well compress-

*Max Planck Institute for Informatics, Saarland Informatics Campus (SIC), Saarbrücken, Germany
{kbringma, marvin, wellnitz}@mpi-inf.mpg.de
†Saarbrücken Graduate School of Computer Science

[1]By $\tilde{O}$-notation we hide logarithmic factors, that is, $\tilde{O}(T) = \bigcup_{c\ge 0} O(T \log^c T)$.

ible, but also for DNA sequencing, where the input is not very well compressible, but its extensive length makes even small space reductions highly desirable. Indeed, in a big data world, since compressed data can be stored more efficiently, can be transmitted using less resources such as energy and bandwidth, and sometimes can even be processed faster, there is barely any reason *not* to compress our data.

Lossless compression schemes range from Kolmogorov complexity (perfect compression, intractable to compute or even approximate) to run-length encoding (very efficient computation, barely any compression on natural texts). Practically useful compression schemes like the Lempel-Ziv family are somewhere in between these two extremes, since an optimal compression can be computed or at least approximated efficiently, and they yield good compression rates in practice. In this paper, we consider *grammar compression*, a notion that has proven to be influential for algorithm design, since it is mathematically elegant while being equivalent, up to lower order terms (moderate constants and logarithmic factors), to popular compression schemes such as members of the Lempel-Ziv family [33, 47, 44], byte-pair encoding [42], dictionary methods, and others [37, 35]. Moreover, grammar compression is fairly generic and thus likely to capture future compression schemes.

A grammar compression of a string t is a context-free grammar generating the language $\{t\}$. Equivalently, we can consider Straight Line Programs (SLPs). An SLP $\mathcal{T}$ is defined over an alphabet Σ and consists of a sequence of non-terminals $T_1, \ldots, T_n$. Each non-terminal is associated with a replacement rule: Either T_i represents an alphabet symbol $\sigma \in \Sigma$, in which case we write $T_i \to \sigma$. Or T_i represents a concatenation of two previous non-terminals $T_\ell, T_r, \ell, r < i$, in which case we write $T_i \to T_\ell T_r$. The string represented by T_n is the string compressed by the SLP. See Section 2 for a more formal definition and an example.

To learn more about grammar compressions, we refer the reader to the works [46, 32, 18, 41, 25, 38, 39, 36, 40]. For a variety of string problems it is known that on grammar-compressed input they can be solved faster than simply decompressing the input and then running a classic (uncompressed) algorithm. For example, exact pattern matching on a text with SLP-size n and a pattern of length m can be solved in near-linear time $\tilde{O}(n+m)$ [27][2]. Several researchers have also studied exact pattern matching on LZW-compressed strings [3, 29, 20, 21, 19]; this task is closely related.

[2] Note that [27] actually considers a different setting in which *both* text and pattern are given in compressed form – here, we simply state the resulting running time for our setting.

For pattern matching with k mismatches, the typical situation studied in the literature is that the text has very large uncompressed text length N but is given compressed by an SLP of size n (or in LZW-compressed form of size n), while the pattern of length m is given in uncompressed form. In particular, on SLPs the problem can be solved in times $O(n(\min\{mk, k^4+m\}+\log N))$ [7] and $O(nm \log m)$ [43], and on LZW-compressed text it can be solved in time $O(n\sqrt{m}k^2)$ [23, 6].

Note that this running time is far from linear even for very small k, in contrast to exact pattern matching, which can be solved in near-linear time in the input size $n + m$ [27]. This leaves the following question.

Is pattern matching with mismatches on compressed text in near-linear time for small k?

Our main algorithmic result is an affirmative answer to this question.

THEOREM 1.1. *Pattern matching with k mismatches on a text t given by an SLP of size n and a pattern p of length m can be solved in time $O(n\, k^3\, (k \log k + \log m) + k\, m)$.*

We obtain this running time both for deciding whether a k-match exists and for counting the number of k-matches. We also show how to enumerate all k-matches in time $O(nk^3(k \log k + \log m) + km + occ)$, where occ is the number of occurrences. The running time is near-linear in the input size $n + m$ (plus the output size in the case of enumerating) for constant or slightly superconstant k. We leave it as an open problem to further improve the dependence on k. Our algorithmic advancement was made possible by a structural insight into pattern matching with mismatches, which we describe next.

Structure of Pattern Matching with Mism. For exact pattern matching, that is, the case $k = 0$, the structure of matches is fairly simple. Consider a pattern p of length m and an (uncompressed) text t of length $n \leq \frac{3}{2}m$. Consider the substring t' of t from the beginning of the leftmost exact match of p to the end of the rightmost exact match of p. Then the classic periodicity lemma [16] (see also e.g. [20, Lemma 2.4]) implies that p and t' are periodic strings with the same period x, and the starting positions of exact matches are all offsets of p in t' matching exactly the period x. In other words, the following structural characterization holds. (Here we denote the infinite repetition of a string z by z^*, the length of z by $|z|$, and the substring of z from its i-th to its j-th position by $z[i, j]$.)

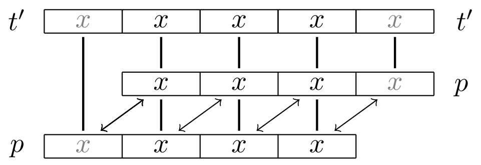

Figure 1: The structure of matches without mismatches is easy: The parts labeled x must be equal as p matches in t only if all letters of p match without a mismatch. (Implied equalities are denoted by an arrow.)

Fact 1.1. *Given strings p of length m and t of length at most $\frac{3}{2}m$, let t' denote the shortest substring of t such that any exact match of p in t is also an exact match in t'. If there are at least two matches of p in t, then there is a substring x of p such that $p = x^*[1,m]$ and $t' = x^*[1,|t'|]$, that is, the pattern and the interesting part of the text are periodic with period x.*

Moreover, the exact matches of p in t' start at positions $1+i\cdot|x|$ with $0 \le i \le (|t'|-|p|)/|x|$. (Consider Figure 1 for a visualization.)

So exact pattern matching is well understood. In contrast, no structural characterization of pattern matching with mismatches is known, leaving us with the following question.

What is the solution structure of pattern matching with k mismatches?

To answer this question, let us start with some examples. In all our examples, the pattern has length m, the text has length $2m$, and the alphabet consists of symbols $\{a, b\}$.

(1) For text $t = a^m b^m$ and pattern $p = a^{m/2}b^{m/2}$, there is one exact match, and all shifts of this exact match of up to $\pm k$ yield a k-match. Thus, in this case both the pattern and text are far from periodic but there still exist several k-matches. For this reason, we can only hope to show periodicity when the number of k-matches is at least some polynomial in k.

(2) Start with $t = a^{2m}$ and $p = a^m$ and perturb them by changing $k/2$ random positions to b in both strings. Then with high probability both strings are aperiodic, but every offset yields a k-match. Thus, this is a case with a very large number of k-matches (polynomial in m), but text and pattern are not perfectly periodic. For this reason, we can only hope to show periodicity up to $O(k)$ mismatches, that is, the pattern p has Hamming distance $O(k)$ from a periodic string.

(3) Start with $t = a^{2m}$ and $p = a^m$ (with $3\,|\,m$), change the $(\frac{1}{3}\cdot 2m)$-th and $(\frac{2}{3}\cdot 2m)$-th positions in t to b, and change the middle $k+1$ positions in p to b. That is, $t = a^{2m/3-1}\,b\,a^{2m/3-1}\,b\,a^{2m/3}$ and $p = a^{(m-k-1)/2}\,b^{k+1}\,a^{(m-k-1)/2}$. Then for $m \gg k$, any offset matches at most one pair of b's in t and p. Any offset aligning a pair of b's in t and p yields a k-match, while all remaining offsets have more than k mismatches. Therefore, the set of starting positions of k-matches can be written as a union of two intervals. In particular, the k-matches *do not form an arithmetic progression*. This is in contrast to Fact 1.1, where the starting positions $1+i\cdot|x|$ form an arithmetic progression in t' (and thus also in t).

In the example, we can remedy this fact by introducing an "approximate" notion: We say that an arithmetic progression A *approximates* the set of k-matches if every k-match starts at some position $i \in A$, and every starting position $i \in A$ yields an $O(k)$-match. In this sense, the trivial arithmetic progression $A = \{1, 2, \ldots, m+1\}$ approximates the k-matches of p in t in the example.

These examples yield necessary relaxations for a structural result of pattern matching with mismatches. We show that they are also sufficient, by proving the following structural result. (Here we denote the Hamming distance by δ_H.)

Theorem 1.2. *Given strings p of length m and t of length at most $2m$, at least one of the following holds:*

- *The number of k-matches of p in t is at most $O(k^2)$.*
- *Let t' denote the shortest substring of t such that any k-match of p in t is also a k-match in t'. Then there is a substring x of p, with $|x| \le m/k$, such that $\delta_H(p, x^*[1,m]) \le O(k)$ and $\delta_H(t', x^*[1,|t'|]) \le O(k)$. Moreover, any k-match of p in t' starts at a position of the form $1+i\cdot|x|$ with $0 \le i \le (|t'|-|p|)/|x|$ (but not every starting position $1+i\cdot|x|$ necessarily yields a k-match).*

The first case covers example (1). In the second case, the Hamming distance $O(k)$ to a periodic string covers example (2). Further, note that in the second case every starting position $1+i\cdot|x|$ of p in t' yields an $O(k)$-match, and thus we obtain an approximate characterization of k-matches by an arithmetic progression as in example (3).

We remark that similar, but weaker structural results can be extracted from previous work. First, the previously fastest algorithm for pattern matching with mismatches in compressed text [23] implicitly shows a similar result with the error bounds $O(k^2)$ and $O(k)$ replaced by the trade-off $O(m/z)$ and $O(z\,\mathrm{poly}(k))$, via studying so-called z-breaks in the pattern. Note that at least one of these error bounds is $\Omega(\sqrt{m})$. Second, the fastest streaming algorithm for pattern matching with mismatches shows a result [13, Lemma 4.3] that is superficially related, although it is not clear whether a specific structural result like Theorem 1.2 could be extracted – even if it could, the error bound would be at least $\Omega(\mathrm{poly}(k)\log m)$ (and it is not clear to the authors what the polynomial in k would be). Our result yields error bounds that are independent of m and only depend on k.

We conjecture that the first case can be improved to $O(k)$ (note that example (1) only rules out a bound of $o(k)$), and we leave it as an open problem to prove such a tight structural result.

Technical Overview For our structural result, we start by splitting the pattern p into $200k$ parts of roughly equal size, and note that in any k-match at least $199k$ of these parts must be matched exactly. Fix one of these parts p_i, and denote its period by x_i. Now we enumerate, starting in p_i and walking in p to the right, the mismatches with respect to period x_i, and we stop after we have seen $20k$ mismatches or when we reach the end of p. We do the same walk to the left of p_i.

The *periodic case* is when there are less than $20k$ mismatches to the left and to the right of some p_i in p. In this case, p has Hamming distance less than $40k$ to some substring of the infinite repetition x_i^*, that is, $\delta_H(p, x^*[1,m]) \le 40k$ for some period x (which is a cyclic rotation of x_i). By triangle inequality, any k-match of p in t has Hamming distance less than $41k$ to $x^*[1,m]$. Note that t' can be covered by two k-matches (the leftmost and the rightmost) by the assumption that the inequality $n \le 2m$ holds for the text and pattern length. We therefore obtain $\delta_H(t', x^*[1,|t'|]) \le 82k$, so both p and t' are almost periodic with period x. Finally, we argue that since there are many repetitions of x in p, and thus in each k-match most of them must be matched without any mismatches, at least one x in p and t' must match exactly (and thus all x's match exactly).

We turn to the *aperiodic case*, in which for all p_i we have at least $20k$ mismatches to the left or to the right of p_i with respect to x_i^*. Without loss of generality say that there are $20k$ mismatches to the right of p_i. In the text, consider all maximal substrings $t_1, \dots, t_\ell$ with period x_i. By iterating over all p_i, we can assume that p_i is exactly matched, and thus we can use Fact 1.1 to infer that p_i must be matched into some substring t_j. We may ignore the substrings t_j of length less than $|p_i|$, in particular we only need to consider $O(k)$ such substrings. We now do a similar walking to before to enumerate mismatches with respect to x_i^* to the right of t_j in t, stopping after we see $19k$ mismatches or once we reach the end. Then we can show that at least $18k$ of the $20k$ mismatches that we enumerated in p must be matched among the $19k$ mismatches that we enumerated in t. This leaves $O(k^2)$ possible matches. Since we also have a choice over the part p_i in the pattern and the substring t_j in the text, we obtain an upper bound of $O(k^4)$ on the number of k-matches. We improve this bound to $O(k^2)$ by using a marking trick twice. This finishes the proof overview for the structural result.

To prove our algorithmic result (Theorem 1.1) we borrow some basic algorithmic tools as well as a reduction to so-called pattern-compressed-strings from [23]. In the remainder of the algorithm, we essentially follow the above proof of our structural result in order to find a k-match. In particular, we also split into $\Theta(k)$ parts, and for each part use an efficient solution to enumerate $\Theta(k)$ mismatches with respect to a periodic string in time $O(k)$. This leaves us in either the periodic or aperiodic case, which get special treatment similar to the above arguments.

Related Work Many hardness results are known for compressed strings. For example, computing the Hamming distance between two strings given by SLPs is known to be #P-complete [34], and computing the smallest Hamming distance of any substring of t to p is known to take time $\min\{N, nm\}^{1\pm o(1)}$ for a text of length N and SLP-size n and a pattern of length m, over constant alphabet size [1]. However, these bounds do not apply to the setting that we study in this paper, where k is constant or slightly superconstant, and indeed we present a near-linear time algorithm in this setting.

A problem closely related to pattern matching with mismatches is pattern matching with *errors*, where instead of a substring of the text with small Hamming distance to the pattern, we are looking for a substring with small *edit distance* to the pattern. This problem has also been extensively studied, both in the uncompressed (see, e.g., [30, 14]) and the compressed setting (see, e.g., [23]).

2 Preliminaries

We denote the set $\{1, \dots, n\}$ by $[n]$.

Strings For a string s we denote its length by $|s|$. For strings s, s', we write $s' \preceq s$ if s' is a substring of s, and if s' is also shorter than s, we write $s' \prec s$. Further, we write $s[i, \dots], 1 \le i \le |s|$, to denote the suffix of s starting at position i. Similarly, we use $s[i,j], 1 \le i,j \le |s|$, to denote the substring of s that starts at position i and ends at position j. In particular, we have $s = s[1, |s|]$. Whenever we deal with substrings algorithmically, we store them as pairs of start and end position. We denote the empty string by ε.

We write s^R to denote the reversed string of s. Further, let $s \cdot s'$ denote the concatenation of the strings s and s'. For an integer $c \geq 1$, we write s^c for the string consisting of c repetitions of s; we call s^c a *power* of c. We use s^* to denote the string consisting of an infinite number of repetitions of s.

The *period* of s is the shortest prefix p of s such that $s \prec p^*$. If p is the period of s and $|p| \leq |s|/2$, we call s periodic (with period p), otherwise we say that s is aperiodic. Further, for a string s and a string $\hat{p} = p^c, c > 0$ where p is aperiodic, a *power stretch* of $\hat{p}$ in s is a maximal substring $s' \preceq s$ that can be written in the form $s' = p^d, d \geq c$. Consider also the following example of power stretches of `abab` in the string `ababaaababab`.

$$\underline{\texttt{a b a b}}\ \texttt{a a}\ \underline{\texttt{a b a b a b}}$$

Hamming Distance and Common Substrings Given two strings t and t' of equal length, we write $\delta_H(t, t')$ to denote their *Hamming distance*, which is defined as the number of positions where t and t' differ. Formally, we have

$$\delta_H(t, t') := |\{i \in [|t'|] \mid t[i] \neq t'[i]\}|.$$

We refer to the positions where t and t' differ as *mismatches.*

Further, we say that a string p *matches* in another (possibly longer) string t at position $i \in [|t| - |p| + 1]$ with k mismatches, if

$$\delta_H(p, t[i, i + |p| - 1]) \leq k.$$

In this case, we also say that there is a *k-match* of p in t at position i (we will drop the k if it is understood from the context). Additionally, if we want to emphasize that a string matches without any mismatches, we say that it matches *exactly.*

Given two strings t and t', the *longest common prefix (suffix)* of them is the longest string that is a prefix (suffix) of both t and t'. For any fixed string, the following result shows how to compute longest common prefixes and suffixes between its substrings fast.

Fact 2.1. (`LCSuf`, `LCPref`) *Let a string s of length m be given. After preprocessing s for $O(m)$ time, given any substrings $t, t' \preceq s$, we can compute the longest common prefix (suffix) of t and t' in constant time.*

We will call these procedures `LCSuf` *and* `LCPref` *in the remainder.*

Fact 2.2. (Substring Concatenation Queries) *Let a string s of length m be given. After preprocessing s for $O(m)$ time, given any substrings $t, t' \preceq s$, we can locate $t \cdot t'$ in s if it is a substring of s or report that $t \cdot t' \not\preceq s$ in $O(\log \log m)$ time. This problem is also called* Substring Concatenation Query Problem *in the literature, see e.g. [8, 15, 26, 45].*

Straight-Line Programs (SLPs) A *Straight-Line Program* or SLP is a context-free grammar that generates exactly one string. Equivalently, we may think of an SLP as a set of non-terminals $\{T_1, \dots, T_n\}$ and productions of the form $T_i \to \sigma$ or $T_i \to T_\ell T_r$, where $\ell, r < i$. We identify SLPs with their sets of non-terminals and write $\mathcal{T} = \{T_1, \dots, T_n\}$ to denote an SLP. One can obtain the string $\text{eval}(T_i)$ represented by a non-terminal T_i as follows:

$$\text{eval}(T_i) := \begin{cases} \sigma, & \text{if } T_i \to \sigma \\ \text{eval}(T_\ell) \cdot \text{eval}(T_r), & \text{if } T_i \to T_\ell T_r. \end{cases}$$

The string represented by $\mathcal{T}$ is obtained via $\text{eval}(\mathcal{T}) = \text{eval}(T_n)$. We write $|T_i|$ for $|\text{eval}(T_i)|$. Consider Figure 2 for an example of an SLP.

Given an SLP $\mathcal{T}$, a string p and an integer k, we are interested in solving the following problems:

- determine whether there exists a k-match of p in $\text{eval}(\mathcal{T})$ (DecideMatch),
- count the number of k-matches of p in $\text{eval}(\mathcal{T})$ (CountMatches), and
- compute all k-matches of p in $\text{eval}(\mathcal{T})$ (EnumerateMatches).

3 Structure of Pattern Matching with Mism.

In this section we present our new structural insight, that is, we bound the number of possible matches for far-from-periodic strings. Specifically, we prove the following theorem, which is the same as Theorem 1.2, but with explicit constants.

Theorem 3.1. *Given strings p of length m and t of length at most $2m$, at least one of the following holds:*

- *The number of k-matches of p in t is at most $627k^2$.*
- *Let $t' \preceq t$ denote the shortest substring of t such that any k-match of p in t is also a k-match in t'. Then there is a string $x \prec p$, $|x| \leq \lceil m/18k \rceil$, such that $\delta_H(p, x^*[1, m]) \leq 6k$ and $\delta_H(t', x^*[1, |t'|]) \leq 14k$, that is, the pattern and the match-containing part of the text are* almost *periodic. Moreover, any k-match of p in t' starts at a position of the form $1 + i \cdot |x|$ with $0 \leq i \leq (|t'| - |p|)/|x|$ (but not every starting position $1 + i \cdot |x|$ necessarily yields a k-match).*

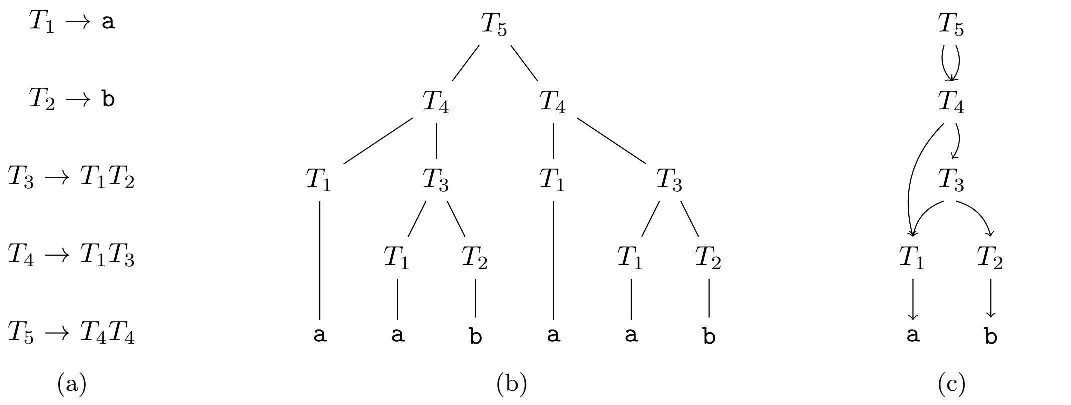

Figure 2: (a) An SLP generating aabaab. (b) The corresponding parse tree. (c) The corresponding acyclic graph.

Proof. Partition p into $18k$ parts $p_1, \dots, p_{18k}$ of (almost) equal length. (If m is not a multiple of $18k$, we partition p in such a way that the lengths of different parts differ by at most one.) Any match of p in t must match at least $17k$ parts exactly, that is, without any mismatch. Fix a part $p_i = p[a, b]$ and let x_i denote its period, that is, $x_i = p[a, a+|x_i|]$. We let $\hat{x}_i = p[a, b']$ denote the longest substring of p_i that is a power of x_i. Note that $\hat{x}_i$ is almost as long as p_i, that is, $\lfloor m/36k \rfloor \le |\hat{x}_i| \le \lceil m/18k \rceil$.

Consider the mismatches between $p[b'+1, \dots]$ and x_i^* (or, more precisely, between $p[b'+1, m]$ and $x_i^*[1, m-b']$). If there are more than $3k$ mismatches, consider only the first $3k$ of them. Similarly, consider the mismatches between $(p[1, a-1])^R$ and $(x_i^R)^*$, and keep only the first $3k$ if there are more. We do a case distinction on the number of these mismatches.

Case 1: For some i, there are less than $3k$ mismatches before and after p_i. In this case, the pattern is almost periodic. Specifically, p has Hamming distance less than $6k$ from some substring of x_i^*. Equivalently, for some cyclic rotation x of x_i we have $\delta_H(p, x^*[1, m]) < 6k$.

Consider the shortest substring $t' \preceq t$ such that any k-match of p in t is also a k-match in t', and let $n := |t'|$. We may assume that t' is non-empty, as otherwise there is no k-match of p in t and we are done. We will show that all starting positions of k-matches of p in t differ by a multiple of $|x|$. To this end, we first show the following claim.

CLAIM 3.1. *Consider two k-matches of p in t', namely $t'[e, e+m-1]$ and $t'[f, f+m-1]$, with $f > e$. If $f-e$ is not a multiple of $|x|$, then we have $f \ge e+m-16k|x|$.*

Proof. Consider the substrings $x^*[f-e+1, m]$ and $x^*[1, e+m-f]$ of p that are aligned to the same substring of t' under the two different matches. Consider Figure 3 for a visualization.

By triangle inequality, we have

$$\begin{aligned}
&\delta_H(x^*[f-e+1, m], x^*[1, e+m-f]) \\
&\quad \le \delta_H(x^*[f-e+1, m], p[f-e+1, m]) \\
&\qquad + \delta_H(p[f-e+1, m], t'[f, e+m-1]) \\
&\qquad + \delta_H(t'[f, e+m-1], p[1, e+m-f]) \\
&\qquad + \delta_H(p[1, e+m-f], x^*[1, e+m-f]).
\end{aligned}$$

By replacing strings by superstrings, we can further bound

$$\begin{aligned}
&\delta_H(x^*[f-e+1, m], x^*[1, e+m-f]) \\
&\quad \le \delta_H(x^*[1, m], p[1, m]) \\
&\qquad + \delta_H(p[1, m], t'[e, e+m-1]) \\
&\qquad + \delta_H(t'[f, f+m-1], p[1, m]) \\
&\qquad + \delta_H(p[1, m], x^*[1, m]).
\end{aligned}$$

Since $t'[e, e+m-1]$ and $t'[f, f+m-1]$ are k-matches, and since $\delta_H(p, x^*[1, m]) \le 6k$ (by the assumption of Case 1), we obtain

$$\delta_H(x^*[f-e+1, m], x^*[1, e+m-f]) \le 14k. \tag{3.1}$$

Now assume for the sake of contradiction that $f < e+m-16k|x|$. Then the strings $x^*[f-e+1, m]$ and $x^*[1, e+m-f]$ have length at least $16k|x|$. In particular, they both contain at least $15k$ full repetitions of x. However, as x is aperiodic, aligning x with a shifted version of itself yields at least one mismatch. Hence, since $f-e$ is not a multiple of $|x|$, we obtain

$$\delta_H(x^*[f-e+1, m], x^*[1, e+m-f]) \ge 15k.$$

This contradicts Inequality 3.1 and finishes the proof of the claim.

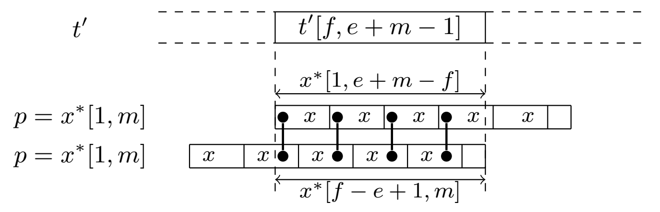

Figure 3: Two overlapping k-matches with misaligning x's. Filled dots mark positions of mismatches in t to either k-match of p.

By Claim 3.1, two k-matches of p that misalign the x's have to lie at least $m - 16k|x|$ apart. As $16k|x| \leq \frac{16}{18}m$, there can be at most 9 matches of p in t that misalign the x's.

Now consider that there are exactly two k-matches that misalign the x's. Those two matches cover at least $2m - 16k|x|$ characters of the text. In the remaining part of the text, a k-match (aligning the x's) may occur every $|x|$ characters. Hence, in this case the number of matches is at most $2 + 16k$.

As $m - \frac{16}{18}m \geq \lceil m/18k \rceil \geq |x|$, this completes the proof in the case of matches with misaligning x's.

Hence, we may assume that the starting positions of all k-matches of p in t differ by an integer multiple of $|x|$. Note that the leftmost k-match is $t'[1,m]$. Using the triangle inequality on $\delta_H(t'[1,m], p) \leq k$ and $\delta_H(p, x^*[1,m]) < 6k$, we obtain

$$\delta_H(t'[1,m], x^*[1,m]) < 7k.$$

Similarly, the last k-match is $t'[n-m+1, n]$, and we have

$$\delta_H(t'[n-m+1, n], x^*[1,m]) < 7k.$$

Note that t' is covered by the first and the last k-match, since the text length is at most $2m$. Since the starting positions of these k-matches differ by integer multiples of $|x|$ as well, this yields

$$\delta_H(t', x^*[1,n]) < 14k.$$

In total, we showed that $\delta_H(p, x^*[1,m]) \leq 6k$ and $\delta_H(t', x^*[1,|t'|]) \leq 14k$. Moreover, every k-match of p in t' aligns the x's, so it starts at a position $1 + c \cdot |x|$, for $0 \leq c \leq (|t'| - m)/|x|$. This concludes the proof in Case 1.

Case 2: For all i, on at least one side of p_i at least $3k$ mismatches to x_i^* exist. Assume without loss of generality that the first $3k$ mismatches to x_i^* *after* p_i exist.

Any k-match of p in t that matches p_i exactly also has to match $\hat{x}_i$ exactly. Thus, for every k-match of p in t that matches p_i exactly, there exists a corresponding substring of t that is equal to $\hat{x}_i$. So consider all different power stretches $t_1, \ldots, t_\ell$ of $\hat{x}_i$ in t. As $\hat{x}_i$ is *long*, that is $|\hat{x}_i| \geq \lfloor m/36k \rfloor$, there are at most $148k$ different power stretches: Indeed, any position in t is contained in at most two different power stretches of length at least $\lfloor m/36k \rfloor \geq m/(37k)$, and thus this number is bounded by $(2 \cdot 2m)/m \cdot 37k \leq 148k$.

Consider any k-match exactly matching p_i inside some $t_j = t[c,d]$. Using the triangle inequality, at least $3k - k = 2k$ mismatches must exist between $t[d+1, \ldots]$ and x_i^*. Call the positions of the first $2k$ mismatches $d < \tau_1 < \cdots < \tau_{2k}$:

$$\forall u \in [2k] : \; t[\tau_u] \neq x_i^*[\tau_u - d].$$

Similarly, consider only the first $2k$ mismatches in p. Call the positions of these mismatches $b' < \pi_1 < \cdots < \pi_{2k}$:

$$\forall u \in [2k] : \; p[\pi_u] \neq x_i^*[\pi_u - b'].$$

In particular, consider an alignment of p to t that exactly matches $\hat{x}_i$ to a substring $t'_j = t[c', d'] \preceq t_j$. Note that $t[d'+1, d]$ is, by definition, equal to the first $d - d'$ letters of x_i^*. This in particular means that the mismatches τ_u between $t[d+1, \ldots]$ and x_i^* are also the first $2k$ mismatches between $t[d'+1, \ldots]$ and x_i^*. (Therefore, it suffices to argue about $t[d+1, \ldots]$ and the mismatches between that part of t and x_i^*, that is, we only need to consider the case $d' = d$.)

Observe that among the positions π_u and τ_v, at least k must align in any k-match. Indeed, otherwise since there are more than k mismatches π_u in p not aligned to any $\tau_1, \ldots, \tau_{2k}$, we would need to align at least one of them to some mismatch in t after τ_{2k}. However, then among the mismatches $\tau_1, \ldots, \tau_{2k}$, there remain more than k unaligned mismatches that we cannot match to any mismatches in p after π_{2k}. Thus, there would be more than k unaligned mismatches, which cannot happen in a k-match.

This observation already bounds the number of possible k-matches of p in t by $O(k^4)$: For every match, there are $18k$ choices for a p_i that matches exactly, at most $148k$ choices for a corresponding t_j, and for each pair of p_i and t_j at least one of the $2k$ mismatches between p and x_i^* has to align to one of the $2k$ mismatches between t and x_i^*, resulting in at most $4k^2$ possible k-matches for that pair of p_i and t_j. Hence, in total in Case 2, there are at most $10656k^4$ possible k-matches of p in t.

We now improve the bound on the number of possible k-matches by applying a marking trick twice, in each step tightening the bound by a factor of k.

First, as before, fix a part p_i in p and a corresponding power stretch $t_j = t[c,d]$ in t. Also, let

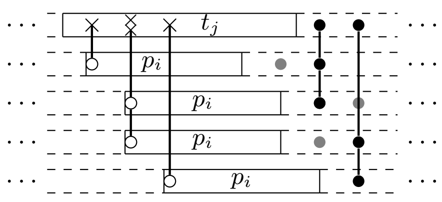

Figure 4: Positions involved in the marking phase. Mismatches that get aligned are denoted by a black dot, the position we mark is denoted by the symbol "$\times$". Note that we mark the first position twice.

$b' < \pi_1 < \cdots < \pi_{2k}$ denote the positions of the first $2k$ mismatches between p and x_i^* and let $d < \tau_1 < \cdots < \tau_{2k}$ denote the positions of the first $2k$ mismatches between t and x_i^*. For each position π_u in p and τ_v in t, put a mark at position $\tau_v - \pi_u + a$ in t. Consider Figure 4 for a visualization of the marking phase. By our previous observation, we know that any k-match of p in t must align the character $p[a]$ to a character in t that has at least k marks, as only these positions may correspond to alignments where at least k of the considered mismatches in p and in t align. As there are only $4k^2$ marks in total, only $4k$ positions in t may have at least k marks.

Second, for every position α in t obtained by the above step for $p_i = p[a, b]$ and a corresponding stretch t_j, mark t at position $t[\alpha - a + 1]$ with a new mark. As every k-match of p in t must match $17k$ parts p_i exactly, only at positions in the text with at least $17k$ new marks a k-match is possible. In total, since there are $18k$ choices for p_i, $144k$ choices for a corresponding t_j and at most $6k$ candidate positions, at most $18k \cdot 148k \cdot 4k = 10656k^3$ positions are marked in the text, thus at most $10656k^3/17k \le 627k^2$ positions are possible starting points of a k-match.

All in all, we proved that either p and t have small Hamming distance to a (highly) periodic string, or the number of k-matches of p in t is at most $627k^2$. This concludes the proof.

4 Algorithm: From Straight-Line Programs to Pattern-Compressed Strings

In the remainder of the paper we show how to apply Theorem 3.1 to obtain a faster algorithm for finding matches with mismatches if the text is given as a straight-line program, proving Theorem 1.1. We use so-called *pattern-compressed strings* (pc-strings for short) introduced by [23] as an intermediate step, that is, we first show how to reduce our problem to the corresponding one for pc-strings in this section. In the next section, we present our new, faster algorithm for solving the problem for pc-strings.

We start by stating the definition of pc-strings and known tools to operate on them.

Definition 4.1. (Pattern-Compressed String [23]) *Let p be a string of length m. We call a string $f = v_1 \dots v_q$ a* p-pattern-compressed string, *in short p-pc-string, if and only if $v_i \preceq p$ for all $i \in [q]$.*

We call f a minimal p-pc-string *if it also satisfies $v_i v_{i+1} \not\preceq p$ for all $i \in [q-1]$.*

We represent such a string as a list $(a_1, b_1), (a_2, b_2), \dots, (a_q, b_q)$, where $v_i = p[a_i, b_i]$. We call the v_i's the factors *of f. We omit p if it is understood from the context.*

Note that our definition is slightly different from the one given in [23], where a pc-string is a minimal pc-string that additionally satisfies the bounds $|f| = \sum_{i\in[q]} |v_i| \le 2|p|$ and $q \le 4k+5$, where k is an additional parameter. Note further that our notion of minimal p-pc-strings coincides with that of *text covers* from [4].

Proposition 4.1. (Compare Lemma 8 in [23]) *After preprocessing a pattern p of length m for time $O(m)$, given two minimal p-pc-strings $f = v_1 \dots v_q$ and $f' = u_1 \dots u_r$, we can construct a new minimal p-pc-string f'' representing the string $f \cdot f'$ in time $O(q + r + \log\log m)$. We denote this operation by $f'' = f \cdot_{pc} f'$.*

Proof. Note that the string $v_1 \dots v_q u_1 \dots u_r$ is almost the desired result, except that $v_q \cdot u_1$ may be a substring of p, in which case we want to merge the two factors. We check whether this is the case by using Fact 2.2 a single time. If $v_q \cdot u_1$ is indeed a substring of p, we create a new factor representing $v_q \cdot u_1$ by setting $v' := p[i, i+|t\cdot t'|-1]$, where i is the index returned by Fact 2.2. Note that as $v_{q-1} \cdot v_q \not\preceq p$, the string $v_{q-1} \cdot v_q \cdot u_1$ cannot be a substring of p, neither can $v_q \cdot u_1 \cdot u_2$.

As in [23], we want to make the technical assumption that every character in the text also appears in the pattern. Our algorithms also work in the general case after slight modifications.

Similar to DecideMatch, CountMatches, and EnumerateMatches, we can define corresponding problems DecideMatchPC, CountMatchesPC, and EnumerateMatchesPC for (minimal) pc-strings, where the text is given as a pc-string instead of an SLP. The following theorem relates the problems on SLPs to the corresponding problems on pc-strings.

Theorem 4.1. *Fix a pattern p of length m. Assume that* DecideMatchPC, CountMatchesPC, *and* EnumerateMatchesPC *can be solved on minimal p-pc-strings of length $O(m)$ consisting of $O(k)$ factors in times $T_D(m,k)$, $T_C(m,k)$, and $T_E(m,k) + O(occ)$, respectively, where occ is the number of k-matches.*

Then the corresponding problems on SLPs (of size n) can be solved in times $O(n \cdot (T_D(m,k) + k + \log\log m) + m)$, $O(n \cdot (T_C(m,k) + k + \log\log m) + m)$, and $O(n \cdot (T_E(m,k) + k + \log\log m) + m + occ)$, respectively.

Proof. Let $\mathcal{T} = \{T_1 \dots T_n\}$ denote the given SLP. For a character $\sigma \in \Sigma$, let $pos(\sigma)$ denote the position of its first occurrence in p. Note that, by our technical assumption, $pos(\sigma)$ exists for every letter $\sigma \in \text{eval}(T_n)$.

Given a non-terminal T_i of $\mathcal{T}$, we define the following minimal pc-strings.

- $pc_{end}[i]$, a minimal p-pc-string of length at most m, consisting of up to $2k+3$ factors, representing any suffix of $\text{eval}(T_i)$. If its length is smaller than m or it has less than $2k+3$ factors, it must represent $\text{eval}(T_i)$.
- $pc_{start}[i]$, a minimal p-pc-string of length at most m, consisting of up to $2k+3$ factors, representing a prefix of $\text{eval}(T_i)$. If its length is smaller than m or it has less than $2k+3$ factors, it must represent $\text{eval}(T_i)$.
- if $T_i \to T_\ell T_r$, we set $pc_{mid}[i] = pc'_{end}[\ell] \cdot_{pc} pc'_{start}[r]$ to be a minimal pc-string where $pc'_{end}[\ell]$ and $pc'_{start}[r]$ are a suffix of $pc_{end}[\ell]$ and a prefix of $pc_{start}[r]$, respectively, consisting of at most $2k+2$ factors each. If either $pc'_{end}[\ell]$ or $pc'_{start}[r]$ has less than $2k+2$ factors, it is equal to $pc_{end}[\ell]$ or $pc_{start}[r]$, respectively.

Claim 4.1. *Let T_i denote a non-terminal of $\mathcal{T}$.*

- *The pc-string $pc_{start}[i]$ represents a suffix of $\text{eval}(T_i)$ that contains every suffix of $\text{eval}(T_i)$ that has a Hamming distance of at most k to some proper prefix of p.*
- *The pc-string $pc_{end}[i]$ represents a prefix of $\text{eval}(T_i)$ that contains every prefix of $\text{eval}(T_i)$ that has a Hamming distance of at most k to some proper suffix of p.*
- *If $T_i \to T_\ell T_r$, the pc-string $pc_{mid}[i]$ contains all k-matches that start in $\text{eval}(T_\ell)$ and end in $\text{eval}(T_r)$.*

Proof. As two consecutive factors of a minimal pc-string together do not form a substring of the pattern p, matching a part of p to a p-pc-string (with at most k mismatches) may cover at most $2k+3$ consecutive factors of that pc-string. (Note that a match might not start at the beginning or end at the end of a factor.)

For $pc_{mid}[i]$, thus any match of p that starts in $\text{eval}(T_\ell)$ and ends in $\text{eval}(T_r)$ may span at most $2k+3-1$ of the factors of $pc_{end}[\ell]$ from its end and at most $2k+3-1$ factors of $pc_{start}[r]$ from its beginning, as it must span the first character of $pc_{start}[r]$ and the last character of $pc_{end}[\ell]$.

The other claimed results follow immediately.

Claim 4.2. *For every k-match of p in $\mathcal{T}$ there is a k-match of p in exactly one $pc_{mid}[i]$ and vice-versa.*

Proof. As the pc-strings pc_{mid} represent substrings of $\mathcal{T}$, any match found in one of the pc-strings is also a match in $\mathcal{T}$.

For the other direction, consider a substring $s \preceq \text{eval}(\mathcal{T})$ with $\delta_H(s,p) \leq k$. Now let $T_i \to T_\ell T_r$ denote the non-terminal of $\mathcal{T}$ that generates the smallest string containing s. Note that this is the only non-terminal that s "crosses" in the sense that s can be split into $s = s_{pre} \cdot s_{suf}$ such that $\text{eval}(T_\ell) = a \cdot s_{pre}$ and $\text{eval}(T_r) = s_{suf} \cdot a'$ (for some strings a, a').

Further, by Claim 4.1, there is a k-match of p in the pc-string $pc_{mid}[i]$. This concludes the proof.

Note that Claim 4.2 implies that matches found in different pc-strings are also different matches in the original SLP. Hence, we can combine the results obtained by oracle calls for the different pc_{mid} in the straight-forward way using a dynamic programming approach on the structure of the SLP: For a non-terminal $T_i \to T_\ell T_r$, the number of all k-matches of p in T_i is the same as the sum of the number of all k-matches in T_ℓ and T_r separately plus the number of match found in $pc_{mid}[i]$. As by definition $\ell, r < i$, we can compute the number of all k-matches of T_i for every i in increasing order starting from $i = 1$. Similar approaches work for DecideMatch and EnumerateMatch.

It remains to show how to construct the pc-strings pc_{mid}. As constructing these pc-strings in a naive way would be too slow, we demonstrate a simple dynamic programming algorithm. Note that it suffices to compute pc_{start} and pc_{end} for each non-terminal, as the corresponding pc_{mid} can be obtained from them via using $\cdot_{pc}$ (possibly up to three times, as we may need to use $\cdot_{pc}$ to obtain the correct sub-pc-strings of the corresponding pc_{start} and pc_{end}).

The actual construction of pc_{end} and pc_{start} itself is straight-forward. For a non-terminal $T_i \to \sigma \in \Sigma$, we set pc_{start} and pc_{end} to $[(pos(\sigma), pos(\sigma))]$ which obviously satisfies all required properties.

For a non-terminal $T_i \to T_\ell T_r$, assume that we already computed $pc_{start}[\ell]$ and $pc_{start}[r]$. If $pc_{start}[r]$

represents $\text{eval}(T_\ell)$ completely, we set $pc_{start}[i]$ to a prefix of $pc_{start}[\ell] \cdot_{pc} pc_{start}[r]$ that represents either $\text{eval}(T_i)[1,m]$ or consists of up to $4k+5$ factors, whichever is shorter. It is easy to verify that this choice indeed fulfills all required properties.

If $pc_{start}[r]$ represents a proper prefix of $\text{eval}(T_\ell)$, the choice $pc_{start}[i] = pc_{start}[r]$ already yields the desired pc-string.

The computation of $pc_{end}[i]$ is symmetric.

For the running time, by Proposition 4.1, a single concatenation operation on pc-strings $\cdot_{pc}$ takes time $O(\log\log m)$. As we use $\cdot_{pc}$ for each non-terminal of $\mathcal{T}$ at most a constant number of times, the transformation of $\mathcal{T}$ into $O(n)$ minimal pc-strings takes time $O(nk + n\log\log m)$. Further, we need to preprocess the pattern to obtain positions of characters and to prepare efficient concatenation of pc-strings. In total, we obtain a running time of $O(nk + n\log\log m + m)$.

As the sets of found matches are disjoint and also for $T_i \to T_\ell T_r$, any match in T_ℓ is located before any matches in T_i and that any match in T_i is located before any match in T_r, we can report all matches in time $O(n + occ)$ by using lists for representing the sets. Then we can perform a union operation on the sets in constant time, as we only need to link the lists. Further, the addition of a constant can be implemented by adding special elements to the list marking the beginning and the end of the range to be incremented. After we build this list for T_n, we traverse it once to resolve all special elements placed in the list in time $O(n + occ)$.

For counting the number of matches, for each non-terminal $T_i \to T_\ell T_r$, we add the number of matches obtained for T_ℓ and T_r to the number of matches obtained from the pc-string corresponding to T_i. The decision problem is even more straight-forward.

As we generate minimal pc-strings of length $O(m)$ consisting of $O(k)$ factors, we can solve the given problem on our pc-strings in time $O(T_D(m,k))$, $O(T_C(m,k))$, or $O(T_E(m,k) + occ)$, respectively, where occ is the number of k-matches. In total, we obtain

- algorithms for detecting and counting k-matches of a pattern p in an SLP $\mathcal{T}$ running in time $O(n \cdot (T_D(m,k) + k + \log m) + m)$ and $O(n \cdot (T_C(m,k) + k + \log m) + m)$, respectively, and
- an algorithm for enumerating all k-matches of a pattern p in an SLP $\mathcal{T}$ running in time $O(n \cdot (T_E(m,k) + k + \log m) + m + occ)$.

5 Faster Algorithm for PC-Strings

Combining the algorithm from [23] for pattern matching with k mismatches in pc-strings with Theorem 4.1, we directly obtain an algorithm for the corresponding problem in SLPs that runs in time $O(n\sqrt{m}k^2 + m)$. We improve this result by improving the running time of the algorithm for pc-strings. We start by introducing some additional tools to operate on pc-strings, in particular, we demonstrate tools to implement the main operations as suggested by Theorem 3.1: To this end, we show how to compute mismatches between a pc-string and a periodic string in Section 5.1 and how to find power stretches of a given string in a pc-string in Section 5.2. Note that while our algorithms are used on different pc-strings, the pattern is always the same. Thus we can preprocess the pattern to obtain speed-ups in the actual algorithms. Finally, we give our main algorithm in Section 5.3.

5.1 Computing Mismatches to Periodic Strings

To compute mismatches between a pc-string and a periodic string, we need to find parts where both strings are equal. In particular, using Fact 2.1, we can obtain fast algorithms to compute the longest common prefix and suffix between a p-pc-string f and the periodic continuation of a substring x of the pattern p.

Lemma 5.1. (`LCPrefPC`, `LCSufPC`) *After preprocessing a pattern p of size m for time $O(m)$, given a p-pc-string $f = v_1 \dots v_q$ and strings x_{suf} and x_{pref}, where $x := x_{pref} \cdot x_{suf} \preceq p$, we can compute the longest common prefix (suffix) of f and $(x_{suf} \cdot x_{pref})^*$ in time $O(|f|/|x| + q)$.*

Proof. As both x_{suf} and x_{pref} and the factors of f are substrings of p, we can compute their longest common prefix in constant time (after a linear preprocessing on p) using Fact 2.1. Using this fact, starting with the empty prefix, we try to extend the common prefix by successively matching factors of f against x_{suf} (or x_{pref}, respectively). The only point where we need to be careful is if such a "jump" would cross the border between two factors of f. In that case, only a part of x_{pref} or x_{suf} gets matched.

If only a part of x_{suf}, say $x_{suf}[1,i]$, got matched, we need to match the part $x_{suf}[i+1,\dots]$ next. We can achieve this by using $x_{pref} := x_{pref} \cdot x_{suf}[1,i]$ and $x_{suf}[i+1,\dots]$ for our next computation on $v_2 \dots v_q$. As $x_{pref} \cdot x_{suf} = x_{pref} \cdot x_{suf}[1,i] \cdot x_{suf}[i+1,\dots] \preceq p$, the new x_{pref} is still a substring of p, so we can still compute prefixes between p and the new x_{suf} and x_{pref} in constant time.

If, however, x_{suf} got matched completely but only a part of x_{pref}, say $x_{pref}[1,i]$ got matched, we need to match the part $x_{pref}[i+1,\dots]$ next. Again, we can achieve this by setting x_{pref} and x_{suf} appropriately for our next computation, namely to $x_{suf} := x_{pref}[i,\dots] \cdot x_{suf}$ and $x_{pref} := x_{pref}[1,i]$. Again, as $x_{pref} \cdot x_{suf} =$

```
LCPrefPC($f = v_1 \dots v_q$, $x_{pref}$, $x_{suf}$)
    if $q = 0$ then return $0$;
    $cur \leftarrow$ LCPref($v_1$, $x_{suf}$);
    if $cur = |v_1|$ then
        return $cur +$ LCPrefPC($v_2 \dots v_q$, $x_{pref} \cdot x_{suf}[1, cur]$, $x_{suf}[cur+1, \dots]$);
    if $cur < |x_{suf}|$ then
        return $cur$;
    $cur \leftarrow cur +$ LCPref($v_1[cur+1, \dots]$, $x_{pref}$);
    if $cur = |v_1|$ then
        return $cur +$ LCPrefPC($v_2 \dots v_q$, $x_{pref}[1, cur - |x_{suf}|]$, $x_{pref}[cur - |x_{suf}| + 1, \dots] \cdot x_{suf}$);
    if $cur < |x_{pref} \cdot x_{suf}|$ then
        return $cur$;
    return LCPrefPC($v_1[cur+1, \dots]\, v_2 \dots v_q$, $x_{pref}$, $x_{suf}$);
```

Algorithm 1: Computing the longest common prefix of a pc-string and a periodic string

$x_{pref}[1, i] \cdot x_{pref}[i+1, \dots] \cdot x_{suf} \preceq p$, we can continue using Fact 2.1. Note that we cannot shift the parts of x_{suf} and x_{pref} as before, as the string $x_{suf} \cdot x_{pref}$ is not necessarily a substring of the pattern.

If at any point, we cannot match x_{pref} or x_{suf} completely, a mismatch occurred and we found the longest common prefix.

If we matched both x_{suf} and x_{pref} completely, we again need to match x_{suf} next. As we do not require our pc-strings to be minimal, we can use a recursive call with a shortened first factor of f (which is still a substring of p).

Consider Algorithm 1 for an implementation.

For the running time $T(f, |x|)$, we group the calls to `LCPrefPC` according to the number of remaining factors of f. Within the first group, we shorten the first factor v_1 of f by length $|x|$ in each recursive call. Hence, we call `LCPrefPC` at most $O(|v_1|/|x|+1)$ times. As each call to `LCPref` takes constant time, we obtain the following recurrence:

$$T(f = v_1 \dots v_q, |x|) \leq O(1+|v_1|/|x|) + T(v_2, \dots, v_q, |x|),$$

which resolves to

$$T(f = v_1 \dots v_q, |x|) \leq O(q + |f|/|x|).$$

Using `LCSuf` instead of `LCPref` and starting with v_q, we obtain a similar algorithm for computing the longest common suffix, which we will call `LCSufPC`. This concludes the proof.

Computing the longest common prefix of two strings also computes the first position where they differ, namely the character directly after the longest common prefix. We exploit this fact to compute the first ℓ mismatches between a pc-string and the periodic continuation of another string or report that there are less than ℓ mismatches between the strings.

Lemma 5.2. (`Mismatches`) *After preprocessing a pattern p of size m for time $O(m)$, given a p-pc-string $f = v_1 \dots v_q$, a substring x of p, and an integer ℓ, we can find the first ℓ mismatches (or report that there are less than ℓ mismatches) between f and x^* (symmetrically, between f^R and $(x^*)^R$) in time $O(q + \ell + |f|/|x|)$.*

Proof. Using Lemma 5.1, we can find the longest common prefix of (a substring of) f and x^*. At the next position both strings must differ (or f got completely matched). Note that we did not necessarily match a power of x, so we need to take care with the next start of the prefix computation. Since Lemma 5.1 allows us to split x into two parts, this is just a minor technicality. Thus, we may continue to compute longest common prefixes until we either found ℓ mismatches or f got matched completely.

Formally, consider Algorithm 2.

```
Mismatches($f = v_1 \dots v_q$, $x$, $\ell$)
    $errs \leftarrow []$; $rem \leftarrow f$; $pos \leftarrow 0$;
    $x_{suf} \leftarrow x$; $x_{pref} \leftarrow \varepsilon$;
    while $|rem| > 0$ and $|errs| \leq \ell$ do
        $len \leftarrow$ LCPrefPC($rem$, $x_{pref}$, $x_{suf}$);
        if $len = |rem|$ then break ;
        $pos \leftarrow pos + len + 1$;
        $errs \leftarrow errs + [pos]$;
        $rem \leftarrow f[pos+1, \dots]$;
        $x_{pref} \leftarrow x[1, pos \bmod |x|]$;
        $x_{suf} \leftarrow x[1 + pos \bmod |x|, \dots]$;
    return $errs$;
```

Algorithm 2: Computing mismatches between a pc-string and a periodic string

For the correctness of Algorithm 2, observe that the counter `pos` always corresponds to the position of the last character that was found to be a mismatch. Thus, the part $x[1, pos \bmod |x|]$ corresponds to the part of x whose suffix is equal to f directly before the mismatch at position `pos`. Therefore, to obtain the next mismatch after the one at position `pos`, we need to find the longest common prefix of $f[pos+1,\dots]$ and $(x[1+pos \bmod |x|,\dots] \cdot x[1, pos \bmod |x|])^*$, which is exactly what the next call to `LCPrefPC` computes.

Bounding the running time naively would yield a bound of $O(\ell\cdot(|f|/|x|+q))$, for using the algorithm from Lemma 5.1 at most ℓ times. However, note that the bound from Lemma 5.1 is only reached if f is traversed completely, but as that algorithm stops as soon as a mismatch is found, `Mismatches` traverses f *in total* at most once. Thus for bounding the running time of `Mismatches`, we may consider all (at most ℓ) calls to `LCPrefPC` *together* as a single call to `LCPrefPC`. As the remaining parts of the algorithm take constant time each, we can bound the running time of `Mismatches` by $O(q+\ell+|f|/|x|)$.

In total, we obtain an algorithm to find the first ℓ mismatches between a pc-string $f = v_1 \dots v_q$ and a periodic string x^* in time $O(q+\ell+|f|/|x|)$.

To obtain an algorithm to compute the first mismatches between f^R and $(x^*)^R$, use `LCSufPC` on f and x^*.

We will later use Lemma 5.2 to find the first $O(k)$ mismatches of a string x of length $\Theta(m/k)$ in a pc-compressed string of $O(k)$ factors, as required by Theorem 3.1, resulting in a running time of $O(k + m/(m/k)) = O(k)$.

A variation of the idea of the previous lemmas can be used to prove the following useful tool.

Proposition 5.1. (`Verify`, [23, Proposition 1]) *After preprocessing a pattern p of length m for $O(m)$ time, given a position i in a p-pc-string $f = v_1 \dots v_q$, we can determine whether aligning the pattern p at that position yields less than ℓ mismatches in time $O(q+\ell)$. That is, we can check whether $\delta_H(f[i, i+|p|-1], p) \le \ell$ in time $O(q+\ell)$.*

Proof. Instead of partitioning a given x into a prefix and a suffix, we operate directly on p instead. That is, we repeatedly find the longest common prefix between the current factor of f and the current suffix of the pattern. Both the implementation and the analysis are very similar to Lemma 5.2 and are thus skipped here.

We will call this procedure `Verify` in the remainder.

We will use `Verify` to check whether a k-match starts in a pc-string consisting of $O(k)$ factors at a given position in time $O(k)$. This concludes the presentation of tools related to finding mismatches between a pc-string and a periodic string.

5.2 Computing Power Stretches In this part we derive an algorithm `Stretches` that finds all power stretches of a string $x \prec p$ in a p-pc-string. We proceed in two steps: First, we give an algorithm `FirstMatch` that locates the first match of x in the pc-string using a modified version of the algorithm in [21]. In a second step, we show how to extend such a match as far as possible using `LCPrefPC` from Lemma 5.1.

Implementing `FirstMatch`. For technical reasons, we adapt an algorithm for multiple-pattern matching in LZW compressed strings [21] for our purposes. Note that in [23, Lemma 11] the algorithm from [21] is adapted to work for a slightly different problem. Unfortunately, we cannot use that adaption directly.

Instead, we follow the lines of [21] to find a string $x \prec p$ in a p-pc-string f. First, we state a convenient tool of [21] that we will use as a subroutine.

Lemma 5.3. ([21, Theorem 1]) *After preprocessing a collection of patterns $p_1,\dots,p_\ell$, $\sum_{i\in[\ell]} |p_i| = m$ for $O(m)$ time, given any sequence S of q substrings $s_1 \preceq p_{i_1},\dots,s_q \preceq p_{i_q}$, we can find the first match of one of the patterns p_i in S in time $O(q \log m)$.*

Corollary 5.1. (`SnippetMatch`) *After preprocessing a string x of size m for $O(m)$ time, given any x-pc-string $f = v_1 \dots v_q$, we can find the first match of x in f in time $O(q \log m)$.*

Proof. Set $\ell = 1$ and $p_\ell = x$ in Lemma 5.3.

We will call this algorithm `SnippetMatch` in the remainder.

To further follow the outline of the algorithm in [21], we need to implement the following operations:

- Given a factor v of f, compute its longest suffix (prefix) that is a prefix (suffix) of x.
- Check whether a factor v of f is fully contained in x, and locate that subword if it is.
- Find the first occurence of x completely contained in a factor v of f.

As we shall see later, the third operation can be solved by a precomputation on p – thus we will focus on the remaining two operations. Using a tool for suffix trees (Fact 5.1), we obtain an algorithm for the first operation in Proposition 5.2. An algorithm for the second operation is then presented in Proposition 5.3.

Fact 5.1. ([22]) *A (reversed) suffix tree of a string s of length m can be preprocessed in time $O(m)$ such that, given any substring $s[i,j]$, we can find its (implicit or explicit) node in the (reversed) suffix tree in time $O(1)$.*

Using Fact 5.1, we obtain an algorithm for computing longest prefixes (suffixes) of factors of a pc-string that are also suffixes (prefixes) of a string x.

Proposition 5.2. (`LCPrefSuf`, `LCSufPref`) *Compare [20, Lemma 2.2].*

After preprocessing a pattern p of length m and a substring $x \preceq p$ for time $O(m)$, given a substring $v \preceq p$, we can find the longest prefix (suffix) of v that is a suffix (prefix) of x in time $O(1)$.

Proof. Build the suffix tree S for $p\|x\$$, where $\|$ and $\$$ are special symbols not appearing in p. Mark all parent nodes of leaves whose root-to-leaf path does not contain the symbol $\|$. By construction, all marked nodes are explicit nodes in S.

Using a linear scan over S, compute for every explicit node w the lowest marked node on the path from w to the root of S. Afterwards, do the preprocessing for Fact 5.1.

Now given any string $v \preceq p \prec p\|x\$$, we first obtain its (explicit or implicit) node w in S and the lowest explicit node w' above w. Looking up the lowest marked node on the path from w' to the root of S in our precomputed information, we directly obtain the longest prefix of v that is a suffix of x.

In total, this lookup takes time $O(1)$ for finding the node in the suffix tree.

We will call this procedure `LCPrefSuf` in the remainder. Finding the longest suffix of v that is a prefix of x can be achieved by using the reversed suffix tree; which we will call `LCSufPref`.

To prepare our adaptation of the algorithm in [21], it remains to show how to locate a factor of a pc-string f in a given text x.

Proposition 5.3. (`Locate`, [23]) *After preprocessing a pattern p of length m and a string $x \prec p$ for time $O(m)$, given any substring $v \prec p$, we can locate v in x or report that v is not a substring of x in time $O(\log m)$.*

Proof. Compute the suffix array of x and the required data structures for Fact 2.1. Now given a string $v \prec p$, use a binary search on the suffix array to find the longest common prefix of v and any suffix of x. If this prefix has length $|v|$, then return the start position of the corresponding suffix of x; otherwise return $v \not\preceq x$.

We will call this procedure `Locate` in the remainder.

Given the tools presented above, we are ready to combine them (following [21]) to obtain the algorithm `FirstMatch`.

Corollary 5.2. (`FirstMatch`) *After preprocessing strings p of size m and $x \prec p$ for $O(m)$ time, given a p-pc-string $f = v_1 \dots v_q$, we can find the leftmost exact match of x in f in time $O(q \log m)$.*

Proof. We consider two different kinds of matches, those completely contained in a single factor of f (“internal matches”) and those that span at least two factors of f (“crossing matches”). We find the leftmost internal and crossing match (if they exist) and return the leftmost among the up to two obtained matches.

To obtain the leftmost internal match, we determine all matches of x in p (using e.g. [28]) in the preprocessing step. Then, we search for the first factor that has a internal match. To achieve this, we only need a single look-up in our precomputed information per factor of f to obtain the leftmost internal match in that factor (if it exists).

For crossing matches of x, we transform f into (multiple) x-pc-strings and then use Corollary 5.1 to obtain the leftmost match in these x-pc-strings. In particular, we want to generate x-pc-strings such that any crossing match of x in f is also a match in one of the x-pc-strings and vice versa. For that, note that any crossing match of x itself is a x-pc-string. Thus it suffices to construct maximal x-pc-strings from f starting with a suffix of a factor of f that is a prefix of x, then extending for a maximal number of factors of f that are substrings of x, and ending with a prefix of a factor of f which is a suffix of x. We can use `LCSufPref`, `Locate`, and `LCPrefSuf` (from Lemma 5.2 and Proposition 5.3) to construct such a maximal x-pc-string.

Consult Algorithm 3 for the details of the algorithm.

For the correctness it suffices to show that any crossing match of x in f can be found in one of the generated x-pc-strings and vice versa. As the x-pc-strings represent substrings of f, one direction is clear. For the other direction, consider a crossing match of x in f. As f restricted to the match is a x-pc-string, there is a maximal x-pc-string f' that contains x. As by construction, a factor of f can appear in two different x-pc-strings only if it is not a substring of x, f' is even unique.

For the running time, the preprocessing time required for the various tools is $O(m)$, and thus this is also the time required for the preprocessing of this algorithm. Constructing the x-pc-strings takes time $O(\log m)$ per factor, as we call `LCPrefSuf`, `LCSufPref`, and `Locate` at most once for each factor of f. As all generated x-

```
FirstMatch($f = v_1 \dots v_n$, $x$)
    if $\exists$ first factor $v_i$ with $x \preceq v_i$ then
        $first\_internal \leftarrow |v_1 \dots v_{i-1}| + \min\{j \mid v_i[j, j + |x| - 1] = x\}$;
    else
        $first\_internal \leftarrow |f| + 1$;
    $first\_crossing \leftarrow |f| + 1$; $pc \leftarrow []$; $start \leftarrow 1$;
    for $i = 1$ up to $n$ do
        if $pc \neq []$ then
            if $v_i \preceq x$ then  $pc \leftarrow pc + [(a, b)]$, where $v_i = x[a, b]$;
            else
                $cur \leftarrow$ LCPrefSuf($v_i$, $x$);
                $pc \leftarrow pc + [(|x| - cur + 1, |x|)]$;
                $mat \leftarrow$ SnippetMatch($pc$, $x$);
                if  $mat \neq$ "no match found" then /* There is a match of $x$ in $pc$ */
                    $first\_crossing \leftarrow start + mat$;
                    break;
                $pc \leftarrow []$;
        if $pc = []$ then
            $cur \leftarrow$ LCSufPref($v_i$, $x$);
            if $cur > 0$ then
                $pc \leftarrow pc + [(1, cur)]$;
                $start \leftarrow |v_1 + \cdots + v_i| - cur + 1$;
    if $pc \neq []$ and $first\_crossing = |f| + 1$ then
        $mat \leftarrow$ SnippetMatch($pc$, $x$);
        if $mat \neq$ "no match found" then
            $first\_crossing \leftarrow start + mat$;
    if $first\_internal = |f| + 1$ and $first\_crossing = |f| + 1$ then
        return "no match found";
    return $\min\{first\_internal, first\_crossing\}$;
```

Algorithm 3: Regular pattern matching in a pc-string

pc-strings together contain at most $2q$ factors, the algorithm from Corollary 5.1 runs in time $O(q \log m)$. Thus in total, we obtain the claimed bound on the running time.

Implementing `Stretches`**.** Using `FirstMatch` which allows us to compute the leftmost match of a string x in a pc-string f, we are ready to show how to compute all power stretches of x in f.

Lemma 5.4. (`Stretches`) *Let p be a string of length m. Further, let x be aperiodic and $\hat{x} = x^c$ for some $c > 0$ with $\hat{x} \preceq p$. We can preprocess p and $\hat{x}$ in time $O(m)$, so that given any p-pc-string we can find all power stretches of $\hat{x}$ in f in time $O(q \cdot |f| \log(m)/|\hat{x}|)$.*

Proof. We find the power stretches of $\hat{x}$ one at a time, where we do two steps for each stretch. First, we locate a position where $\hat{x}$ occurs in f using `FirstMatch` from Lemma 5.2. Second, we extend that match by powers of x as far as possible using `LCPrefPC` from Lemma 5.1. We continue our search at the next position where a match of x in f may occur. As x is aperiodic, we continue $x/2$ positions before the end of the last computed power stretch. Formally, consider Algorithm 4.

For correctness, after we found a match at a position π, we use `LCPrefPC` to extend that match as far as possible. Now there are two cases. If $\hat{x}$ is aperiodic, that is $\hat{x} = x^c, c = 1$, then there are no matches at any position between π and $\pi + |\hat{x}|/2$. Otherwise, $\hat{x} = x^{c-1}x$. As x is aperiodic, the only matches between π and $\pi + (c-1) \cdot |x| + |x|/2$ start at the positions $\pi + i \cdot |x|$. But these matches are part of the same power stretch as the one starting at position π. Thus in any case, we can skip over at least $(c-1) \cdot |x| + |x|/2$ characters after each match.

For the running time, split the time spent by the algorithm into two parts. First, consider the total time spent for calls to `LCPrefPC`. Similar to Lemma 5.2 we can argue that this part in total takes time $O(q + |f|/|\hat{x}|)$. Now consider the remaining parts of an iteration of the `while`-loop. For us, it suffices to bound the running time of every call to `FirstMatch` by $O(q \log m)$. Then the most expensive remaining operation per loop iteration is the call to `FirstMatch`, the remaining part can be implemented in constant time. As there are at most $|f|/(|\hat{x}| - |x|/2) = O(|f|/|\hat{x}|)$ iterations of the loop, we obtain a running time of $O(q \log(m)|f|/|\hat{x}|)$. As `FirstMatch` and `LCPrefPC` operate on the string independently, we can add their running times to obtain the total running time of $O(q \log(m)|f|/|\hat{x}|)$.

```
Stretches($f = v_1 \dots v_q$, $\hat{x} = x^c$)
    $rem \leftarrow f$; $M \leftarrow \{\}$;
    while $|rem| > 0$ do
        $m \leftarrow$ FirstMatch($rem$, $\hat{x}$);
        if $m =$"no match found" then break;
        $i \leftarrow |x| \cdot \lfloor$LCPrefPC($rem[m, \dots]$,$\varepsilon$,$\hat{x}$)$/|x|\rfloor$;
        $M \leftarrow M \cup \{(|f| - |rem| + m, i)\}$;
        $rem \leftarrow rem[m + i - |x|/2]$;
    return $M$;
```

Algorithm 4: Computing power stretches of a string in a pc-string

In our case, we will use Lemma 5.4 on p-pc-strings and (long) parts x of p according to Theorem 3.1. For these strings, the algorithm has a running time of $O(qk \log m)$, as $|x| = \Theta(|f|/k)$.

5.3 Faster Algorithms for PC-Strings Combining all the subroutines, we can now present our new, faster algorithm for pattern matching with mismatches in pc-strings. In particular, we will show how to *count* all k-matches of a given pattern p in a pc-string f.

Theorem 5.1. *After preprocessing a pattern p of length m for time $O(km + k^2)$, given any p-pc-string $f = v_1 \dots v_q$, we can count all k-matches of p in f in time $O(k^2(q \log m + k^2 \log k))$.*

Together with the reduction of Theorem 4.1, we directly obtain a faster algorithm for pattern matching with mismatches in SLPs. Recall that in the reduction, we construct (minimal) pc-strings of length m that consist of at most $4k + 5$ factors each, simplifying the bound on the running time that we obtain.

Theorem 5.2. *Given an SLP $\mathcal{T} = \{T_1, \dots, T_n\}$, a pattern p of length m, and an integer k, we can count all matches with at most k mismatches in the text generated by $\mathcal{T}$ in time $O(nk^3(k \log k + \log m) + km)$.*

Our algorithms extend to returning the first or enumerating all k-matches naturally.

In the remainder of this section, we prove Theorem 5.1. We split the presentation of the algorithm into two parts, depending on whether or not the pattern is almost periodic in the sense of Theorem 3.1. After presenting algorithms for these two cases, we will then combine them and present how to decide which of the two algorithms to run on a given instance.

Recall from Theorem 3.1 that we split the pattern into $18k$ parts $p_1, \dots, p_{18k}$, where

$$p_i := p[1 + (i-1) \cdot \lfloor m/18k \rfloor + \alpha_i, i \cdot \lfloor m/18k \rfloor + \beta_i]$$

and $0 \le \alpha_i < \beta_i < \alpha_{i+1} < 18k$ are integers adjusting the sizes of the some of the parts by one if $18k \nmid m$ in such a way that the sizes of any two parts differ by at most one.

For each part p_i, we compute its period x_i and its "extended period" $\hat{x}_i = x_i^c \preceq p$, where c is maximal. Further, let a_i and b_i denote the positions in p where $\hat{x}_i$ begins and ends, respectively. Next, for each part, we try to find the first $3k$ mismatches between $p[b_i+1, \dots]$ and $\hat{x}_i^*$ and between $p[1, a_i - 1]^R$ and $(\hat{x}_i^*)^R$ using `Mismatches` from Lemma 5.2. Call the sets Π_i and Π_i^R, respectively. If for every part p_i at least one of the sets Π_i and Π_i^R contains $3k$ mismatches, we say that p is not almost periodic. (Consult also Case 2 of Theorem 3.1.) We consider this case first.

Non-Periodic Case. We postpone the presentation of the preprocessing to Theorem 5.1, and will assume that the required data structures have been built for the following lemmas.

Lemma 5.5. (`NonPeriodicMatches`) *Suppose we are given a pattern p of length m and also for each of the $18k$ parts p_i, $p_i := p[1 + (i-1) \cdot m/18k, i \cdot m/18k]$, the period $x_i = p_i[1, |x_i|]$ and extended period $\hat{x}_i$. Assume that for every part p_i at least one of the sets Π_i and Π_i^R contains $3k$ positions of mismatches. Then, given any p-pc-string $f = v_1 \dots v_q$, we can count (and enumerate) all k-matches of p in f in time $O(k^2(q \log m + k^2 \log k))$.*

Proof. We proceed as suggested by the proof of Theorem 3.1. For each part p_i, we find all power stretches of $\hat{x}_i$ in f using `Stretches` from Lemma 5.4. Then, for each obtained stretch (s, ℓ) (starting at position s and extending for ℓ characters), we compute the set T_s of the first $2k$ mismatches between $\hat{x}_i^*$ and $f[s+\ell, \dots]$ and the set T_s^R of the first $2k$ mismatches between $(\hat{x}_i^*)^R$ and $f[1, s-1]^R$. (Note that we do not need to explicitly reverse the strings, as `Mismatches` uses `LCSufPC` on the non-reversed strings in this case.)

Consider the case where Π_i contains $3k$ mismatch positions, the other case is symmetric. By triangle inequality, the set T_s must contain $2k$ mismatches, as

```
Candidates($f$, $(s,\ell)$, $\hat{x}$, $x$, $\Pi$, $\Pi^R$)
    $T_s \leftarrow$ Mismatches($f[s+\ell,\dots]$, $\hat{x}$, $2k$);
    $T_s^R \leftarrow$ Mismatches($(f[1,s-1])^R$, $\hat{x}^R$, $2k$);
    if $|\Pi| = 2k$ and $|T_s| = 2k$ then
        Compute the multiset $R \leftarrow \{s+\ell+\tau-\pi-|\hat{x}| \mid \tau \in T_s, \pi \in \Pi\}$;
        return $\{r \in R \mid r$ appears at least $2k$ times in $R\}$;
    if $|\Pi^R| = 2k$ and $|T_s^R| = 2k$ then
        Compute the multiset $R \leftarrow \{s-\tau+\pi \mid \tau \in T_s^R, \pi \in \Pi^R\}$;
        return $\{r \in R \mid r$ appears at least $2k$ times in $R\}$;
    return $\{\}$;

countNonPeriodicMatches($f = v_1 \dots v_q$, $p$, $k$, $\{(x_i, \hat{x}_i, \Pi_i, \Pi_i^R \mid i \in 18k)\}$)
    for $i = 1$ up to $18k$ do
        $S \leftarrow$ Stretches($f$, $\hat{x}_i$);
        for $(s,\ell)$ in $S$ do
            $R_{i,s} \leftarrow$ Candidates($f$, $(s,l)$, $\hat{x}_i$, $x_i$, $\Pi_i$, $\Pi_i^R$);
        $R_i \leftarrow \{r-(i-1)\cdot m/18k \mid r \in \bigcup_{s\in S} R_{i,s}\}$;
    $R \leftarrow \{r \mid r$ appears in at least $17k$ different $R_i\}$;
    return $|\{r \in R \mid$ Verify($r$)$\}|$;
```

Algorithm 5: Counting k-matches of a not (almost) periodic pattern in a pc-string

otherwise there is no k-match exactly matching p_i to a substring of $f[s, s+\ell-1]$. In particular, for an alignment to result in a k-match, at least $2k - k = k$ mismatches in T_s must get aligned to mismatches in Π_i. We use the "marking trick" presented in Theorem 3.1 to compute all alignments of p_i into $f[s, s+\ell]$ that satisfy this constraint.

Note that the resulting positions in f represent positions where $p_i[1]$ may start. To further operate on these positions, we normalize them by shifting them by $(i-1)m/18k$, such that they mark the positions in f where $p[1]$ needs to be aligned.

Using the normalized positions obtained for every pair of part of p and power stretch in f, we use the "marking trick" again to obtain starting positions where at least $17k$ parts of p get matched exactly. In the end, we use `Verify` from Lemma 5.1 to check for each remaining position whether a k-match is present or not.

Consult Algorithm 5 for the (technical) details. For a visualization of the (first) marking stage of the algorithm, consider Figure 5. Note that as opposed to Theorem 3.1, the positions of the mismatches in Π, Π^R and T, T^R are relative to the part in p and the power stretch, respectively.

For the running time, for each of the $O(k)$ parts of p, we find at most $O(k)$ corresponding power stretches in f in time $O(qk \log m)$. For each pair of part in p and stretch in f, we call `Mismatches` twice to obtain up to $2k$ mismatches each; incurring a running time of $O(q+k)$. Further, we need to compute a sum set of two sets of size $O(k)$ and find all elements of a multiset of size $O(k^2)$

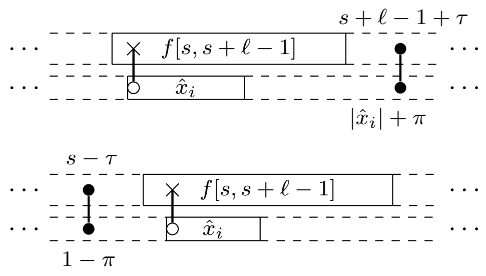

Figure 5: Positions involved in the marking stage of Algorithm 5. Positions in the pattern p are relative to the part p_i. Mismatches that get aligned are denoted by a filled dot, positions we mark are denoted by the symbol "$\times$".

that appear at least k times. Both can be achieved with a total running time of $O(k^2 \log k)$. Determining the positions that appear in sets corresponding to at least k different stretches takes again time $O(k^2 \log k)$. In total, computing the $O(k^2)$ candidate positions takes time $O(k \cdot (qk \log m + k \cdot (q + k + k^2 \log k) + k^2 \log k)) = O(k^2(q \log m + k^2 \log k))$, verifying all positions takes time $O(k \cdot k^2)$, resulting in a total running time of $O(k^2(q \log m + k^2 \log k))$.

Almost Periodic Case. Implementing the case when the pattern is almost periodic turns out to be not as straight-forward as the other case, as we cannot directly obtain the relevant substring f' of f containing

```
countPeriodicMatches($f = v_1 \dots v_q$, $p$, $k$, $x$, $\hat{x}$, $\Pi$)
    $cnt \leftarrow 0$;
    $last \leftarrow -1$;
    $S \leftarrow$ Stretches($f$, $\hat{x}$);
    for $(s, \ell)$ in $S$ in increasing order of $s$ do
        $T \leftarrow$ Mismatches($f[s+\ell, \dots]$, $\hat{x}$, $7k$);
        if $|T| < 7k$ then
            $T \leftarrow T \cup \{m+1\}$;
        $T^R \leftarrow$ Mismatches($(f[1, s-1])^R$, $\hat{x}^R$, $7k$);
        if $|T^R| < 7k$ then
            $T \leftarrow T \cup \{0\}$;
        $E \leftarrow \{\tau + s + \ell - 1 \mid \tau \in T\} \cup \{s - \tau \mid \tau \in T^R\}$;
        sort $E$;
        $W_i \leftarrow E \cap (E[i], E[i+1] + m]$ for all $E[i] \leq |f| - m$;
        for each $W_i$ do
            if $|W_i| + |\Pi| \leq k + 1$ then
                if $E[i] > last$ then
                    $cnt \leftarrow cnt + |\{s + c \cdot |x_i| \mid c \in \mathbb{Z}, E[i] < s - c \cdot |x_i| \leq E[i+1]\}|$;
                    $last \leftarrow E[i+1]$;
            else
                $R \leftarrow \{\tau - \pi + 1 \mid \tau \in W_i, \pi \in \Pi\}$;
                $R \leftarrow \{r \in R \mid r$ has $> |W_i| + |\Pi| - k$ appearances in $R\}$;
                for $r$ in $R$ do
                    if $r > last$ and $E[i] < r \leq E[i+1]$ and Verify($r$) then
                        $cnt \leftarrow cnt + 1$ /* $k$-match at position $r$.                    */
                        $last \leftarrow r$;
    return $cnt$;
```

Algorithm 6: Counting k-matches of an (almost) periodic pattern in a pc-string

all k-matches of p. Instead, we again search for power stretches and handle each stretch separately.

LEMMA 5.6. (PeriodicMatches) *Suppose we are given a pattern p of length m, a string $\hat{x} \prec p$, $\lfloor m/36k \rfloor \leq |\hat{x}| \leq \lceil m/18k \rceil$, satisfying $\delta_H(\hat{x}^*[1,m], p) < 6k$, and a set Π of the positions where $\hat{x}^*[1,m]$ and p differ. Then given any p-pc-string $f = v_1 \dots v_q$, we can count all k-matches of p in f in time $O(k^4 + qk \log m)$.*

Proof. As the pattern has $\mu \leq 6k$ mismatches to $\hat{x}^*[1,m]$, by triangle inequality, any substring f' of f where p has a k-match can have at most $7k$ mismatches to $\hat{x}^*[1,m]$. Hence, if there is a match of p in f, there are two mismatches between f' and $\hat{x}^*[1,m]$ that lie at least $m/7k$ characters apart. As $|\hat{x}| \leq \lceil m/18k \rceil$, the string $\hat{x}$, and thus a power stretch of $\hat{x}$, is thus a substring of t'.

Fix a power stretch (s, ℓ) of $\hat{x}$ in f.

- Let T and T^R denote sets of positions in f, where T contains the positions of the first $7k+1$ mismatches between $\hat{x}_i^*$ and $f[s+\ell, \dots]$. Likewise, T^R contains the positions of the first $7k+1$ mismatches between $(\hat{x}_i^*)^R$ and $f[1, s-1]^R$.

  We can compute the sets T and T^R by using Mismatches from Lemma 5.2 and a constant number of operations to translate the obtained relative positions into absolute positions in f.

- Set $\tau_{min} := \min T^R$ if $|T^R| = 7k+1$ and $\tau_{min} := 0$ otherwise. Further, set $\tau_{max} := \max T$ if $|T| = 7k+1$ and $\tau_{max} := |f|+1$ otherwise.

  By construction and using the triangle inequality, we obtain that any match of p in f that aligns a part of p to $f[s, s+\ell]$ must lie in $f[\tau_{min}+1, \tau_{max}-1]$.

- Let $E := T \cup T^R \cup \{\tau_{min}, \tau_{max}\}$ denote a (sorted) set of *events*. For the i-th event $e_i \in E$ define the i-th *window* $W_i := E \cap (e_i, e_{i+1}+m]$ if $|f| - e_i > m$, otherwise W_i is undefined.

  Note that by construction $|E| \leq 14k + 4$, which is also a bound on the number of different windows.

- For any window W_i we want to obtain all matches of p starting in f at a position after e_i and not after e_{i+1}. We distinguish between the following two cases.

Case 1: The total number of mismatches is small, that is $|W_i| + \mu \leq k$. (Recall that μ is the number of mismatches between the pattern p and $\hat{x}^*$.) By construction, f is almost equal to a substring of $\hat{x}^*$. In particular, we have $f[j] = (\hat{x}_{suf}\hat{x}^d\hat{x}_{pre})[j]$ for a constant $d > 0$, a suffix $\hat{x}_{suf}$ of $\hat{x}$, a proper prefix $\hat{x}_{pre}$ of $\hat{x}$, and $j \in [\min W_i, \max W_i] \setminus W_i$.

In this case, p matches at every position $j \cdot |\hat{x}| + W_i[1] + |\hat{x}_{suf}|$ for $0 \leq j \leq (e_{i+1} - m - e_i - |\hat{x}_{suf}|)/|\hat{x}_{suf}|$, that is, aligning the periods $\hat{x}$ of p and $f[\min W_i, \max W_i]$ always yields a match.

As $|\hat{x}| \leq \lceil m/18k \rceil$, misaligning the periods cannot result in a match, thus we cannot obtain any additional matches by misaligning periods.

Case 2: The total number of mismatches is large, that is $|W_i| + \mu > k$. By triangle inequality, for any alignment of p to $f[\min W_i, \max W_i]$ to result in a match, at least $|W_i| + \mu - k$ positions in W_i must be aligned to a mismatch in p. As there are only at most $O(k^2)$ many different possibilities to align a position in W_i with a mismatch in p, we can try all of them via using `Verify` from Lemma 5.1.

Consider Algorithm 6 for the details. Note that different power stretches in f may have sets W in common. To not overcount the matches, we consider every set W at most once. Traversing the stretches and the sets W in every stretch from left to right, a single counter suffices to keep track of the already processed sets W.

For the running time, note that there are at most $O(k)$ power stretches of $\hat{x}$ in f, which we can find in time $O(qk \log m)$. For every power stretch, we find $O(k)$ mismatches in time $O(q + k)$, after which we use a sliding window with at most $O(k)$ different events. In the worst case, for each event, we have to iterate over the $O(k^2)$ pairs of mismatch in p and in f, resulting in a running time of $O(k^4)$ for filtering the positions we want to verify. However, as there are only $O(k^2)$ such pairs per power stretch in total, and we verify every pair at most once, the `Verify` step takes also $O(k^4)$ time.

In total, we obtain an algorithm to count the number of matches of an almost periodic pattern in a pc-string in time $O(k^4 + qk \log m)$.

Preprocessing and Combination. We can finally give the proof of our main algorithmic result.

Proof. [Proof of Theorem 5.1]
The only thing left to present is the preprocessing, Algorithm 7. We need to compute for each part p_i of p the period x_i (denoted by `Period` in Algorithm 7) and the power stretch $\hat{x}_i$ of x_i in p_i. Both the period and its stretch can be found via a linear scan over p_i, resulting in a running time of $O(m/k)$ for each part p_i.

```
Preprocess($p$)
    $\ell \leftarrow \lfloor |p|/18k \rfloor$;
    Create data structures on $p$ for Fact 2.1;
    Do preprocessing for FirstMatch;
    for $i = 1$ up to $18k$ do
        $p_i \leftarrow p[1 + \ell \cdot (i-1), \ell \cdot i]$;
        $x_i \leftarrow$ Period($p_i$);
        $\hat{x}_i \leftarrow x_i^c \preceq p$, for maximal $c$;
        $\Pi_i \leftarrow$Mismatches($p[\ell \cdot (i-1) + |\hat{x}_i|, \dots]$,
          $\hat{x}_i$, $3k$);
        $\Pi_i^R \leftarrow$ Mismatches($(p[1, \ell \cdot (i-1)])^R$,
          $\hat{x}_i^R$, $3k$);
        if $|\Pi_i| + |\Pi_i^R| < 6k$ then
            return "periodic", $\hat{x}_i$, $x_i$,
              $\{\pi + \ell \cdot (i-1) + |\hat{x}_i| \mid \pi \in \Pi_i\}$
              $\cup \{\ell \cdot (i-1) + 1 - \pi \mid \pi \in \Pi_i^R\}$;
    return "aperiodic", $\{(\hat{x}_i, x_i, \Pi_i, \Pi_i^R) \mid i \in [18k]\}$;
```
Algorithm 7: Preprocessing for finding matches in pc-strings

After having computed all $\hat{x}_i$, we do the preprocessing for the various tools and subroutines we presented earlier. In particular, we will call the preprocessing for Lemma 5.4 with our generated strings $\hat{x}_i$ (for each of them separately), so that we can use it to obtain power stretches of any $\hat{x}_i$ fast. Thus, this step takes time $O(km)$, as we may have up to k different strings $\hat{x}_i$.

Next, for each p_i, we compute the first $4k$ mismatches between p and x_i^* on either side of $\hat{x}_i$ by calling `Mismatches` from Lemma 5.2 using the part of p after (or before) $\hat{x}_i$ as the pc-string. For each part p_i this takes time $O(q + k + k) = O(k)$, as $q = 1$ in this case.

If we happen to find out that p is almost periodic in the sense of Theorem 3.1, we return this information, as well as the period $\hat{x}_i$ and all mismatches to the period. Otherwise, we return for each p_i the computed values.

Using the preprocessing, we can determine whether we use the algorithm for aperiodic patterns from Lemma 5.5 or for (almost) periodic patterns from Lemma 5.6 to obtain the matches in the given pc-string. Combining the running times for both cases, we obtain an algorithm running in time $O(k^2(q \log m + k^2 \log k))$.

References

[1] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 192–203, 2017.

[2] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*, 52(2):299–307, 1996.

[4] A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2), 2007.

[5] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257 – 275, 2004. SODA 2000 Special Issue.

[6] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts. *ACM Trans. Algorithms*, 6(1):3:1–3:14, 2009.

[7] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[8] A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *European symposium on algorithms*, pages 120–131, 2000.

[9] A. Butman, P. Clifford, R. Clifford, M. Jalsenius, N. Lewenstein, B. Porat, E. Porat, and B. Sach. Pattern matching under polynomial transformation. *SIAM J. Comput.*, 42(2):611–633, 2013.

[10] R. Clifford, K. Efremenko, E. Porat, and A. Rothschild. Pattern matching with don't cares and few errors. *J. Comput. Syst. Sci.*, 76(2):115–124, 2010.

[11] R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. A. Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 2039–2052, 2016.

[12] R. Clifford, A. Grønlund, K. G. Larsen, and T. A. Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 22:1–22:14, 2018.

[13] R. Clifford, T. Kociumaka, and E. Porat. The streaming k-mismatch problem. *CoRR*, abs/1708.05223, 2017.

[14] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.

[15] M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140, 1996.

[16] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[17] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[18] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, pages 392–403, 1996.

[19] P. Gawrychowski. Tying up the loose ends in fully LZW-compressed pattern matching. In *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, pages 624–635, 2012.

[20] P. Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25:1–25:17, 2013.

[21] P. Gawrychowski. Simple and efficient LZW-compressed multiple pattern matching. *J. Discrete Algorithms*, 25:34–41, 2014.

[22] P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 455–466, 2014.

[23] P. Gawrychowski and D. Straszak. Beating $\mathcal{O}(nm)$ in approximate LZW-compressed pattern matching. In L. Cai, S.-W. Cheng, and T.-W. Lam, editors, *Algorithms and Computation*, pages 78–88. Springer Berlin Heidelberg, 2013.

[24] P. Gawrychowski and P. Uznanski. Optimal tradeoffs for pattern matching with k mismatches. *CoRR*, abs/1704.01311, 2017.

[25] R. Giancarlo, D. Scaturro, and F. Utro. Textual data compression in computational biology: a synopsis. *Bioinformatics*, 25(13):1575–1586, 2009.

[26] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[27] Jez, Artur. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015.

[28] D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[29] S. R. Kosaraju. Pattern matching in compressed texts. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 349–362. Springer Berlin Heidelberg, 1995.

[30] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 126–136, 1985.

[31] G. M. Landau and U. Vishkin. Efficient string match-

ing with k mismatches. *Theoretical Computer Science*, 43:239 – 249, 1986.

[32] N. J. Larsson. *Structures of string matching and data compression.* Department of Computer Science, Lund University, 1999.

[33] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[34] Y. Lifshits. Processing compressed texts: A tractability border. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, pages 228–240, 2007.

[35] Q. Liu, Y. Yang, C. Chen, J. Bu, Y. Zhang, and X. Ye. RNACompress: Grammar-based compression and informational complexity measurement of RNA secondary structure. *BMC Bioinformatics*, 9(1):176, 2008.

[36] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[37] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

[38] R. Radicioni and A. Bertoni. Grammatical compression: compressed equivalence and other problems. *Discrete Mathematics and Theoretical Computer Science*, 12(4):109, 2010.

[39] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Proc. 31st International Colloquium on Automata, Languages, and Programming (ICALP'04)*, pages 15–27. Springer, 2004.

[40] H. Sakamoto. Grammar compression: Grammatical inference by compression and its application to real data. In *ICGI*, pages 3–20, 2014.

[41] D. Sculley and C. E. Brodley. Compression and machine learning: A new perspective on feature space vectors. In *Proc. Data Compression Conference (DCC'06)*, pages 332–341, 2006.

[42] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte pair encoding: A text compression scheme that accelerates pattern matching. Technical report, Technical Report DOI-TR-161, Department of Informatics, Kyushu University, 1999.

[43] A. Tiskin. Threshold approximate matching in grammar-compressed strings. In *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*, pages 124–138, 2014.

[44] T. Welch. A technique for high-performance data compression. *Computer*, 17:8–19, 1984.

[45] D. E. Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2):81–84, 1983.

[46] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images.* Morgan Kaufmann, 1999.

[47] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.