Space-Efficient k-Mismatch Text Indexes

Tomasz Kociumaka[1] and Jakub Radoszewski[2]

[1]Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany (tomasz.kociumaka@mpi-inf.mpg.de)
[2]Institute of Informatics, University of Warsaw, Warsaw, Poland (jrad@mimuw.edu.pl)

Abstract

A central task in string processing is *text indexing*, where the goal is to preprocess a *text* (a string of length n) into an efficient *index* (a data structure) supporting queries about the text. While the most fundamental *exact pattern matching* queries ask to find all the occurrences of a *pattern* (a string of length m) as substrings of the text, many applications call for *approximate pattern matching* queries, where the pattern may differ slightly from the matching substrings. A breakthrough in the extensive study of approximate text indexing came from Cole, Gottlieb, and Lewenstein (STOC 2004), who proposed *k-errata trees* — a family of text indexes supporting several closely related flavors of approximate pattern matching queries. In particular, k-errata trees yield an elegant solution to *k-mismatch* queries, where the similarity is quantified using an upper bound $k \geq 1$ on the Hamming distance between the pattern and its approximate occurrences. The resulting k-mismatch index uses $\mathcal{O}(n \log^k n)$ space and answers a query for a length-m pattern in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time, where occ is the number of approximate occurrences.

In retrospect, k-errata trees appear very well optimized: even though a large body of work has adapted k-errata trees to various settings throughout the past two decades, the original time-space trade-off for k-mismatch indexing has not been improved in the general case. We present the first such improvement, a k-mismatch index with $\mathcal{O}(n \log^{k-1} n)$ space and the same query time as k-errata trees.

Previously, due to a result of Chan, Lam, Sung, Tam, and Wong (Algorithmica 2010), such an $\mathcal{O}(n \log^{k-1} n)$-size index has been known only for texts over alphabets of constant size $\sigma = \mathcal{O}(1)$. In this setting, however, we obtain an even smaller k-mismatch index of size only $\mathcal{O}(n \log^{k-2+\varepsilon+\frac{2}{k+2-(k \bmod 2)}} n) \subseteq \mathcal{O}(n \log^{k-1.5+\varepsilon} n)$ for $2 \leq k \leq \mathcal{O}(1)$ and any constant $\varepsilon > 0$. Along the way, we also develop improved indexes for short patterns, offering better trade-offs in this practically relevant special case.

1 Introduction

In (full-)text indexing, the goal is to convert a string T of length n (referred to as the *text*) into a data structure (called an *index*) that can efficiently answer subsequent queries about the text T. The most fundamental are *exact pattern matching* queries, which ask to report all occurrences of a given string P of length m (called a *pattern*) as substrings of T. The study of exact text indexes dates back to the early 1970s, when Weiner [Wei73] introduced *suffix trees*. This textbook data structure occupies $\mathcal{O}(n)$ space, supports $\mathcal{O}(n)$-time construction, and answers pattern matching queries in $\mathcal{O}(m + \mathsf{occ})$ time, where occ is the number of occurrences of the pattern P in the text T.[1] Further

[1]The original implementation applies to texts over alphabets of constant size $\sigma = \mathcal{O}(1)$, but modern ones [Far97, FG15] support integer alphabets $\{0, \ldots, \sigma - 1\}$ of any size $\sigma = n^{\mathcal{O}(1)}$. This comes at the cost of increasing the

work in exact text indexing focused mainly on improving the space complexity (the main bottleneck in practice), often at the cost of degraded query time. Selected milestones in this direction include the suffix array [MM93, KSB06], the compressed suffix array [GV05] and the FM-index [FM05], as well as the r-index [GNP20] and other indexes for highly repetitive texts [Nav22].

Another extensively studied direction, motivated by numerous applications [Nav01], is approximate text indexing. An established way to define approximate occurrences of the pattern is through a threshold $k \geq 1$ on the Hamming distance: if $T[i\,..\,i+m)$ and P differ in at most k positions, that is, the two strings have at most k mismatches, then $T[i\,..\,i+m)$ is a *k-mismatch occurrence* of P in T. Given a pattern P, a *k-mismatch index* over T is required to report all the k-mismatch occurrences of P in T. Unlike offline k-mismatch pattern matching algorithms [ALP04, CFP+16, GU18], which operate on a fixed pattern P, an index supports many online queries.

The earliest approximate indexes were designed for the special case of $k = 1$ [AKL+00, BGW00]. These solutions were later optimized [Bel09, Bel15], but the underlying techniques do not generalize to $k > 1$. The seminal contribution to approximate indexing for arbitrary $k \geq 1$ is the *k-errata tree* of Cole, Gottlieb, and Lewenstein [CGL04], which takes $\mathcal{O}(n \log^k n)$ space and lists the k-mismatch occurrences of a length-m pattern in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time.[2] The errata trees have been featured in the *Encyclopedia of Algorithms* [Lew16] and adapted to numerous tasks [BGP+24, CIK+22, CKPR21, GGM+25, GLS18, GHPT25, TAA16, ZLPT24]. Even though several alternative k-mismatch indexes have been presented, none of the known trade-offs improve upon the k-errata trees in the general case. Our $\mathcal{O}(n \log^{k-1} n)$-size index provides the first such improvement for any k: it retains the query time of k-errata trees and supports texts over large alphabets.

Theorem 1.1. *For every text of length n and integer threshold $k \geq 1$, there exists a k-mismatch index of size $\mathcal{O}(n \log^{k-1} n)$ that answers queries in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time, where m is the length of the pattern and occ is the number of k-mismatch occurrences.*

Fifteen years ago, Chan, Lam, Sung, Tam, and Wong [CLS+10] presented an analogous trade-off for texts over constant-size alphabets; for a general alphabet of size σ, the query complexity of their index becomes $\mathcal{O}(\sigma \cdot \log^k n \log\log n + m + \mathsf{occ})$. In this work, we also address the case of $\sigma = \mathcal{O}(1)$: for every constant $k \geq 2$, we present an even smaller k-mismatch index.

Theorem 1.2. *For every text of length n over an alphabet of size $\sigma = \mathcal{O}(1)$, integer threshold $k = \mathcal{O}(1)$, and real constant $\varepsilon > 0$, there exists a k-mismatch index of size $\mathcal{O}(n \log^{k-2+\varepsilon+\frac{2}{k+2}} n)$ for even k or $\mathcal{O}(n \log^{k-2+\varepsilon+\frac{2}{k+1}} n)$ for odd k that answers queries in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time, where m is the length of the pattern and occ is the number of k-mismatch occurrences.*

In particular, our 2-mismatch index is of size $\mathcal{O}(n \log^{0.5+\varepsilon} n)$, that is, $\log^{0.5-\varepsilon} n$ times smaller than the index of [CLS+10] and $\log^{1.5-\varepsilon} n$ times smaller than the 2-errata tree. The savings become more prominent, growing arbitrarily close to $\log n$ and $\log^2 n$, respectively, as k increases. Not only does Theorem 1.1 provide the first general-case improvement in the two-decade-long history of errata trees, but the restricted case of $\sigma = \mathcal{O}(1)$ has also seen no progress since 2010.

Table 1 summarizes our and previously known upper bounds for k-mismatch indexing. We assume the word RAM model with word size $\Omega(\log n)$. Besides the contributions discussed above, earlier work also offers two results that extend the original k-errata tree trade-off in two directions.

construction time to $\mathcal{O}(n \log\log n)$, increasing the query time to $\mathcal{O}(\log\log\sigma + m + \mathsf{occ})$, **or** making the construction randomized.

[2] The k-mismatch index of [CGL04] actually occupies $\mathcal{O}((c^k/k!) \cdot n \log^k n)$ space for a constant $c > 1$. Since we focus on the regime of $k = \mathcal{O}(1)$, in the introduction we omit the $c^k/k!$ factor for simplicity; note that this factor is at most 1 for sufficiently large k. Throughout, log denotes the base-2 logarithm.

Space	**Query time** (plus $\mathcal{O}(m+\mathsf{occ})$)	**Reference**	**Comment**
$\mathcal{O}(n\log^k n)$	$\mathcal{O}(\log^k n \log\log n)$	[CGL04]	improved here
$\mathcal{O}(n\log^{k-1} n)$		[CLS+10]	$\sigma=\mathcal{O}(1)$, improved here
$\mathcal{O}(n\log^{k-1} n)$		**Thm. 1.1**	
$\mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+2-(k \bmod 2)}} n)$		**Thm. 1.2**	$\sigma=\mathcal{O}(1)$, constant $\varepsilon>0$
$\mathcal{O}(n(\alpha\log\alpha\log n)^k)$	$\mathcal{O}((\log_\alpha n)^k \log\log n)$	[Tsu10]	$2\le\alpha\le n/2$
$\mathcal{O}(n\log^{k-h+1} n)$	$\mathcal{O}(\log^{\max(kh,k+h)} n \log\log n)$	[CLS+11]	$1\le h\le k+1$

Table 1: The k-mismatch indexes for arbitrary pattern lengths m, for $k=\mathcal{O}(1)$.

The k-mismatch index by Tsur [Tsu10] allows decreasing the query time at the cost of a larger space. In particular, for constant k and σ, a query time of $\mathcal{O}(\log^k n/\log^k\log n+m+\mathsf{occ})$ can be achieved using $\mathcal{O}(n\log^{k+\varepsilon} n)$ space, for any constant $\varepsilon>0$. Alternatively, $\mathcal{O}(\log\log n+m+\mathsf{occ})$ query time can be obtained using $\mathcal{O}(n^{1+\varepsilon})$ space. A different trade-off, proposed by Chan, Lam, Sung, Tam, and Wong [CLS+11], allows decreasing the space at the cost of slower queries. Among others, they obtain an $\mathcal{O}(n)$-size index with query time $\mathcal{O}(\log^{k(k+1)} n\cdot\log\log n+m+\mathsf{occ})$ and an $\mathcal{O}(n\log^{k-1} n)$-size index with $\mathcal{O}(\log^{2k} n\log\log n+m+\mathsf{occ})$-time queries.

One of the technical contributions behind Theorem 1.2 is a new k-mismatch index optimized for handling short patterns, of lengths polylogarithmic in n. Our and previous solutions relevant for this setting are presented in Table 2. They are all characterized by the product of the index size and query time being $\Omega(nm^{k-1})$. In the query complexity of the indexes from [Cob95, Ukk93], one can trade the $m^{k+1}\sigma^k$ factor by an n factor (which is not compelling for $k=\mathcal{O}(1)$, since offline k-error pattern matching works in $\mathcal{O}(nk)=\mathcal{O}(n)$ time for a string over an integer alphabet [LV89]).

Beyond the worst-case k-mismatch indexes, data structures with good average-case complexity for general k [Al-15, CN21, CO06, EGM+07, GMRS03, Maa06, MN07] as well as for $k=1$ [MN05] were proposed. Solutions that are efficient for particular data have also been studied [CW18, KST16, LLT+09, LD09]. Approximate indexing in external memory was considered in [HLS+11]. Indexes for a small k (e.g., $k\le 7$) were evaluated in practice [CN21, LLT+09].

Space	**Query time** (plus $\mathcal{O}(m+\mathsf{occ})$)	**Reference**	**Comment**
$\mathcal{O}(n)$	$\mathcal{O}(m^{k+2}\sigma^k\log(m+\sigma))$	[Ukk93]	improved by [Cob95]
	$\mathcal{O}(m^{k+2}\sigma^k)$	[Cob95]	
	$\mathcal{O}((\sigma m)^k\log n)$	[HHLS04]	improved by [CLS+10, LSW08]
	$\mathcal{O}((\sigma m)^k\log\log n)$	[LSW08]	
	$\mathcal{O}((\sigma m)^{k-1}\log n\log\log n)$	[CLS+10]	
$\mathcal{O}(n\mu^h\log^2\mu/\log n)$	$\mathcal{O}(m^{k-h}\log m\log\log n)$	**Thm. 1.4**	$m\le\mu=\Omega(\log n)$, $h\in[1\,..\,k)$, $\sigma=\mathcal{O}(1)$
$\mathcal{O}(n\log_\sigma^k n)$	$\mathcal{O}(m^k)$	[Al-15]	

Table 2: The k-mismatch indexes for specific pattern lengths m, for $k=\mathcal{O}(1)$.

On the lower bound side, Cohen-Addad, Feuilloley, and Starikovskaya [CFS19] showed that if a k-mismatch index answers queries in $\mathcal{O}(\log^k n/(2k)^k + m + \mathsf{occ})$ time in the pointer machine model, then it requires $\mathcal{O}(C^k n)$ space for a constant $C > 0$. This result holds for every even k that satisfies $8\sqrt{\log n}/\sqrt{3} \leq k = o(\log n)$. Furthermore, they showed that, under the Strong Exponential Time Hypothesis, no k-mismatch index for $k = \Theta(\log n)$ can be constructed in polynomial time and decide in $\mathcal{O}(n^{1-\delta})$ time (for some constant $\delta > 0$) whether a given pattern of length $m = \Theta(\log n)$ has any k-mismatch occurrence. These results are a step forward in our understanding of the complexity of approximate indexing, but they are still quite far from the complexity of k-errata trees. In particular, no non-trivial lower bounds can be deduced for the basic case of $k = \mathcal{O}(1)$.

1.1 Related indexing problems

A related problem is indexing a text to support queries for patterns with wildcards (i.e., don't care symbols). A significant amount of work has been devoted to this problem [BGVV14, CGL04, LSTY07, LMRT14, LNV14, RI07]. Lower bounds of similar flavor as in [CFS19] for indexing for patterns with wildcards were presented earlier by Afshani and Nielsen [AN16]. The algorithms presented in [BGVV14, CGL04, LMRT14] all employ variants of errata trees. The problem of indexing for patterns with wildcards appears to be easier, in terms of complexity, than approximate text indexing. In particular, the index of Cole, Gottlieb, and Lewenstein [CGL04] occupies $\mathcal{O}(n \log^k n)$ space and answers queries in $\mathcal{O}(2^k \log\log n + m + \mathsf{occ})$ time, that is, with a 2^k rather than a $\log^k n$ factor. For constant k and σ, a trivial linear-space index, already noted in [CGL04], tests all possible substitutions of wildcards in the pattern. This yields $\mathcal{O}(n)$ space (the suffix tree of T) and $\mathcal{O}(m\sigma^k + \mathsf{occ}) = \mathcal{O}(m + \mathsf{occ})$ query time. As a warm-up for the proof of Theorem 1.1, we present the first general improvement over the complexities of [CGL04] in the trade-off for k-errata trees when the pattern contains k wildcards, as stated in the next theorem.

Theorem 1.3. *For a string T of length n, there exists an index using $\mathcal{O}(n \log^{k-1} n)$ space that answers pattern matching queries for patterns of length m with up to k wildcards in $\mathcal{O}(2^k \log\log n + m + \mathsf{occ})$ time.*

A variant of the problem in which the text contains wildcards was also considered [CGL04, HKS+13, LSTY07, TWLY09, Tha13]. Here, for $\sigma, k = \mathcal{O}(1)$, one can fill in all the wildcards in the text in every possible way and then construct a suffix tree for every resulting string text. This leads to $\mathcal{O}(n\sigma^k) = \mathcal{O}(n)$ space and $\mathcal{O}(m + \sigma^k \mathsf{occ}) = \mathcal{O}(m + \mathsf{occ})$-time queries.

Following [CGL04], one can also consider *k-edit indexes*—indexes that report substrings of the text whose edit (Levenshtein) distance to the pattern is at most k. In this work, we focus on the Hamming distance, which is technically simpler: for example, a k-mismatch occurrence of the pattern in the text always has the same length as the pattern, unlike in the case of k-edit occurrences. In particular, all known k-edit indexes can be adapted (and in fact simplified) to operate as k-mismatch indexes.

1.2 Technical overview

Indexing for short patterns with $\sigma, k = \mathcal{O}(1)$

As a stepping stone to Theorem 1.2, we present a k-mismatch index that supports patterns of length $m \leq \mu$, where $\mu = \Omega(\log n)$ is a parameter specified at construction time. Existing linear-space k-mismatch indexes (see Table 2) follow a shared blueprint: exhaustively apply all up to k modifications to the pattern P in every possible way and then perform exact pattern matching

for each variant. This approach yields varying query times depending on how efficiently the exact search is implemented.

Our index, in contrast, balances the cost of these modifications between index size and query time. Suppose we are searching for occurrences of a pattern $P[0\,..\,m)$ with exactly k mismatches. For a trade-off parameter $h \in [1\,..\,k)$, we consider each possible *pivot* $j \in [0\,..\,m)$, aiming to find occurrences in which $P[0\,..\,j)$ has h mismatches and $P[j\,..\,m)$ has $k-h$ mismatches. To support such queries, we preprocess the text T by modifying every suffix $T[i\,..\,n)$ through all possible h substitutions within its prefix $T[i\,..\,i+j)$, and then we build a compact trie containing all resulting modified suffixes. At query time, we modify P in all possible ways by applying $k-h$ substitutions within $P[j\,..\,m)$, and we search for the resulting patterns in the trie. Because this scheme must consider multiple pivots j, care is needed to avoid reporting duplicates, particularly when also handling $(<k)$-mismatch occurrences of P, while ensuring that no additional $\Omega(m \cdot \mathsf{occ})$ query-time overhead is incurred.

We refine the basic scheme through several improvements:

- Naively, locating each modified pattern in the trie takes $\Theta(m)$ time. We reduce this to $\mathcal{O}(\log\log n)$ using a generalization of rooted LCP queries [CGL04] to tries of modified suffixes (see Lemma 3.2 in Section 3).
- Another $\Theta(m)$ factor arises from the need to examine all possible pivots $j \in [0\,..\,m)$, as a given modified pattern may be compatible with up to $\Omega(m)$ pivots. We limit the number of such pivots to $\mathcal{O}(\log m)$ by favoring those divisible by larger powers of two.
- Finally, we save a nearly-$\log n$ factor in space by explicitly storing only the modified suffixes with up to $h-1$ changes. Suffixes with exactly h changes can still be retrieved indirectly via the rank/select data structures of Raman, Raman, and Rao [RRR07].

Together, these refinements yield the following frugal data structure described in Section 4.

Theorem 1.4. *Let T be a string of length n over an integer alphabet of size σ, and let $\mu = \Omega(\log n)$ with $k, \sigma = \mathcal{O}(1)$. For every $h \in [1\,..\,k)$, there exists a k-mismatch index using $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$ space that answers queries for patterns of length $m \le \mu$ in $\mathcal{O}(m^{k-h} \log m \log\log n + \mathsf{occ})$ time.*

Compared to existing data structures for short patterns, ours can be viewed as trading space for improved query time. In particular, for $k, \sigma = \mathcal{O}(1)$ and patterns of length up to $\mu = \mathcal{O}(2^{\log^{1/3} n}) = \mathcal{O}(n^{1/\log^{2/3} n})$, that is, when $\log^3 \mu = \mathcal{O}(\log n)$, the product of space and query time is $\mathcal{O}(n\mu^k \log^3 \mu \log\log n / \log n) = \mathcal{O}(n\mu^k \log\log n)$ matching the corresponding bound of the data structure in [LSW08]. For patterns of length $\mu = \log^{\mathcal{O}(1)} n$, this product in our data structure is actually smaller by a factor of $(\log\log n)^3 / \log n$.

Indexing for long patterns

At a high level, our strategy for patterns of length at least m is to designate $\mathcal{O}(n/m \cdot \mathrm{poly}(k))$ positions in the text T as *anchors* and, at query time, to select $\mathcal{O}(\mathrm{poly}(k))$ anchors in P such that, for every k-mismatch occurrence of P in T, at least one anchor in P coincides with an anchor in T. In general, it is impossible to cover all k-mismatch occurrences in this way, e.g., when $P = \mathrm{a}^m$ and $T = \mathrm{a}^n$. Nonetheless, our anchor selection procedure, which relies on local consistency techniques (string synchronizing sets [KK19]) and the structure of periodic fragments in strings (including Lyndon roots of runs [BII+17]), misses only highly structured k-mismatch occurrences. These can be listed efficiently using a separate subroutine based on interval stabbing [ABR00].

To find typical k-mismatch occurrences, we handle each anchor j in P separately. We first identify all anchors a in T for which $T[a\,..\,n)$ starts with a k-mismatch occurrence of $P[j\,..\,m)$, and all anchors a in T for which $T[0\,..\,a)$ ends with a k-mismatch occurrence of $P[0\,..\,j)$. These two sets are then intersected, taking into account that k is the total mismatch budget across $P[0\,..\,j)$ and $P[j\,..\,m)$. For these steps, we employ two k-errata trees [CGL04] built on a subset of suffixes and reversed prefixes of T, together with an orthogonal range reporting data structure [ABR00, BGW00] constructed on top of them.

This high-level blueprint can be seen as a generalization of the approaches in exact indexing [KK23] (see [KK21, Proposition 6.27]) and [ALP25, Theorem 4] to the k-mismatch setting. Even prior to these works, similar techniques were used for computing the longest common factor with k mismatches [CCI+18, CKPR21] and for circular pattern matching with k mismatches [CKP+21]. Exact [CEK+21, KNO24] and k-mismatch [GHPT25] indexes for highly repetitive texts also share many aspects of this strategy. The key difference is that they exploit compressibility, rather than a lower bound on $|P|$, to reduce the number of anchors in T, and they require additional tools to report so-called secondary occurrences.

We now provide a more detailed overview of our index for long patterns. Suppose the pattern P has length $m \geq (k+1)\gamma$ for some positive integer γ. A prefix of P of length $(k+1)\gamma$ can then be split into $k+1$ substrings $P_1, \ldots, P_{k+1}$ of length γ each, so that every k-mismatch occurrence of P in T contains an *exact* occurrence of at least one of these substrings, say P_i, in T. We compute a synchronizing set in T (cf. [KK19]). In the simpler case where T does not contain substrings that are high-exponent string powers, a τ-synchronizing set for an integer $\tau > 0$ is a subset of $\mathcal{O}(n/\tau)$ positions in T, selected consistently based on the following 2τ characters. The selection is dense: there is a synchronizing position among every τ consecutive positions in T that are not within the final $2\tau - 1$ positions. A length-2τ substring of T starting at a synchronizing position is called a *τ-synchronizing fragment*. Thus, for $\tau = \lfloor(\gamma+1)/3\rfloor$, the occurrence of P_i in T is guaranteed to contain a τ-synchronizing fragment.

A τ-synchronizing set in T does not necessarily extend to a τ-synchronizing set in P. However, since P_i matches a corresponding fragment of T exactly, it must contain a fragment that matches a τ-synchronizing fragment of T; we are interested in the leftmost such fragment $P[j\,..\,j+2\tau)$. We use two ($\leq k$)-errata trees: one built on suffixes of T starting at synchronizing positions, and the other on the complementary reversed prefixes. Here, we exploit the fact that the original k-errata tree, constructed for $x = \mathcal{O}(n/\gamma)$ suffixes of a text, uses only $\mathcal{O}(n + x\log^k x)$ space [CGL04]. For each P_i, we query $P[j\,..\,m)$ in the errata tree of suffixes and $(P[0\,..\,j))^R$ in the errata tree of reversed prefixes, considering all possible distributions of the k mismatches between $P[j\,..\,m)$ and $P[0\,..\,j)$. The results of these two queries are merged into k-mismatch occurrences of the entire P in T using a 2D orthogonal range query over points generated in a manner similar to a tree cross product [BGW00]. To keep the space of the range query data structure small, we prove a new but intuitive bound on the number of occurrences of a given terminal label in an errata tree. Overall, the space complexity is $\mathcal{O}(n\log^{k+\varepsilon} n\,/\,\gamma)$ for any constant $\varepsilon > 0$, where the $\log^{\varepsilon} n$ factor arises from the 2D orthogonal range reporting structure of Alstrup, Brodal, and Rauhe [ABR00]. The query time matches that of a standard errata tree.

If T contains highly periodic substrings, P_i may not include a τ-synchronizing fragment. In this case, either the entire pattern P is nearly periodic, or there are $\Theta(k)$ *misperiods* in P with respect to the period of P_i [CKP+21], and at least one of these misperiods must align with a corresponding misperiod with respect to a run [KK99] of T. In the second case, we can apply the same construction as before, but using misperiods in T and P instead of synchronizing positions. In the nearly periodic case, we design a tailored index based on runs and their Lyndon roots [BII+17]. All nearly periodic substrings of T are grouped according to the run that generates them, the

number of misperiods, and the position where the Lyndon root of the run starts, modulo the period of the run. Subsequently, all runs are grouped by their Lyndon root. For a nearly periodic pattern, we locate all nearly periodic substrings of T at Hamming distance at most k using an interval stabbing data structure. In Section 6, we formalize this approach in the following theorem.

Theorem 1.5. *For a text T of length n, positive integers $k = \mathcal{O}(1)$, $\gamma \in [2\,..\,n]$, and constant $\varepsilon > 0$, there exists an $\mathcal{O}(n + n\log^{k+\varepsilon} n/\gamma)$-space k-mismatch index that answers k-mismatch queries for patterns of length $m \geq (k+1)\gamma$ in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time.*

By combining this approach with the data structure for short patterns, we obtain our k-mismatch index for $k, \sigma = \mathcal{O}(1)$ as stated in Theorem 1.2.

General compact index

A k-errata tree [CGL04] consists of compact tries at levels $0, \ldots, k$. Roughly speaking, each compact trie stores selected suffixes of the text. The compact tries at a higher level are generated from those at the previous level in two different ways, based on nodes and heavy paths, in both cases via so-called group trees.

For a general alphabet of size σ, we save space by explicitly constructing only the $(k-1)$-errata tree in $\mathcal{O}(n\log^{k-1} n)$ space; this corresponds to the compact tries at levels $0, \ldots, k-1$ of the k-errata tree. Chan, Lam, Sung, Tam, and Wong [CLS+10] devised a rather involved approach, based on [RRR07], to avoid generating heavy path-based group trees at level k while still reporting k-mismatch occurrences. Instead of constructing node-based group trees, they used a naive query algorithm, which incurs an extra factor of σ. We show that their technique for heavy path-based group trees can be carefully adapted to node-based group trees as well, effectively eliminating the σ-factor in query time. As a subroutine, we need to report all suffixes of T in a given sorted list that do not contain a specified character at a given position. This can be done by storing an appropriate rank/select data structure of Jacobson [Jac89] over the list of suffixes. Our approach also arguably simplifies that of [CLS+10].

When indexing a text for pattern matching queries with patterns containing $\leq k$ wildcards, heavy path-based group trees are not created. For clarity, in Section 7, we first describe an index of size $\mathcal{O}(n\log^{k-1} n)$ in this setting, achieving the same query time as the corresponding variant of the errata tree.

2 Basics, Strings, Compact Tries

We denote $[i\,..\,j] = [i\,..\,j+1) = \{i, i+1, \ldots, j\}$. A string U is a sequence of characters over a finite alphabet. By $|U|$ we denote the length of U. For every position $i \in [0\,..\,|U|)$, we denote the ith character of U by $U[i]$. A substring of U is a string of the form $U[i] \cdot U[i+1] \cdots U[j-1]$ for integers $0 \leq i \leq j \leq |U|$. A fragment of U is a positioned substring of U, that is, a substring of U together with its specified occurrence in U. By $U[i\,..\,j) = U[i\,..\,j-1]$ we denote the fragment composed of characters $U[i], U[i+1], \ldots, U[j-1]$; if $i = j$, the substring is empty. A fragment $U[i\,..\,j)$ is a prefix of U if $i = 0$ and a suffix if $j = |U|$. By U^R we denote the reversal of U, that is, $U^R = U[|U|-1] \cdots U[0]$. For two strings U and V, by $\mathsf{LCP}(U, V)$ we denote the length of their longest common prefix.

The Hamming distance of two strings U, V of equal length, denoted as $\delta_H(U, V)$, is the total number of positions $i \in [0\,..\,|U|)$ such that $U[i] \neq V[i]$. We say that a pattern P of length m has a k-mismatch occurrence in T at position i if $\delta_H(T[i\,..\,i+m), P) \leq k$.

A string U on which k (at most k) substitutions were performed is called a *k-modified string* U (*$(\leq k)$-modified string* U, respectively). Given a string T, its k-modified fragment can be represented in $\mathcal{O}(k)$ space by storing the original fragment together with the positions of substitutions and the new characters at these positions.

Let us consider an increasing sequence $a_1 < a_2 < \cdots < a_\ell$ composed of integers in $[1\,..\,r]$. A (restricted) *rank* query, given $x \in [1\,..\,r]$, returns $i \in [1\,..\,\ell]$ such that $a_i = x$ or states that no such index i exists. A *select* query, given $i \in [1\,..\,\ell]$, returns a_i.

Theorem 2.1 (Raman, Raman, and Rao [RRR07, Theorem 4.6])**.** *An increasing sequence $a_1 < a_2 < \cdots < a_\ell$ composed of integers in $[1\,..\,r]$ can be represented using $\mathcal{O}(\ell\,\lceil\log(r/\ell)\rceil)$ bits of space*[3] *so that rank and select queries are supported in $\mathcal{O}(1)$ time.*

2.1 Compact Tries

A *trie* $\mathcal{M}$ of a set of strings $\mathcal{S}$ is a rooted tree for which there is a 1-to-1 correspondence between nodes of $\mathcal{M}$ and prefixes of strings in $\mathcal{S}$. There is an edge with label a from node u to node v in $\mathcal{M}$ if and only if u represents a string U and v represents a string Ua for some character a. The root node of $\mathcal{M}$ represents the empty string. Nodes of $\mathcal{M}$ that represent strings from $\mathcal{S}$ are called terminal nodes. For a node v, the concatenation of the labels of edges on the path from the root to v is called the *path label* of v. The *string depth* of a node v is the length of the path label of v. The *locus* of a string U in $\mathcal{M}$, defined only if U is a prefix of a string from $\mathcal{S}$, is the node v in $\mathcal{M}$ with path label U.

A *compact trie* $\mathcal{T}$ of a set of strings $\mathcal{S}$ is a trie of $\mathcal{S}$ in which all non-terminal nodes other than the root with exactly one child are dissolved. Nodes that remain (i.e., the root, branching nodes and terminals) are called explicit, whereas the remaining, implicit nodes can still be addressed indirectly, by specifying their nearest explicit descendant and the length of the path to the descendant in the uncompact trie. By a *node* of $\mathcal{T}$ we mean an explicit or implicit node. Each edge that connects two explicit nodes has a label that is represented as a fragment of one of the strings in $\mathcal{S}$. For every $S \in \mathcal{S}$, we also write $S \in \mathcal{T}$. A compact trie which t terminals contains $\mathcal{O}(t)$ explicit nodes.

The best known example of a compact trie is the suffix tree of the text T, the compact trie of all suffixes of T. We will be considering compact tries of (selected) suffixes of the text T, also called *sparse suffix trees* of T, or of $(\leq k)$-modified suffixes of the text T. In these cases, the label of every edge can be specified in $\mathcal{O}(1)$ or $\mathcal{O}(k)$ space, respectively, as a fragment or a $(\leq k)$-modified fragment of T.

For a length-n text T there is an $\mathcal{O}(n)$-sized data structure [BF00] based on the suffix tree that computes $\mathsf{LCP}(T[i\,..\,n), T[j\,..\,n))$ for any $i, j \in [0\,..\,n)$ in $\mathcal{O}(1)$ time.

2.2 Longest Common Prefix Queries on a Compact Trie

In a $\mathsf{TreeLCP}$ query, we are given a compact trie $\mathcal{T}$ and a pattern P, and we are to report the longest common prefix between P and the strings $S \in \mathcal{T}$. The locus of this longest common prefix is also called the node where P *diverges* from $\mathcal{T}$ and is denoted as $\mathsf{TreeLCP}(\mathcal{T}, P)$.

$\mathsf{TreeLCP}(\mathcal{T}, P)$ queries can be extended by specifying a node v in the compact trie $\mathcal{T}$. Such a query, denoted $\mathsf{TreeLCP}_v(\mathcal{T}, P)$ and referred to as an *unrooted* $\mathsf{TreeLCP}$ query, is equivalent to a

[3] The data structure of [RRR07] actually uses $\lceil\log\binom{r}{\ell}\rceil + o(\ell) + \mathcal{O}(\log\log r)$ bits. If $\ell = r$, then $a_i = i$ for each $i \in [1\,..\,r]$ and rank/select queries can trivially be answered in $\mathcal{O}(1)$ time (a pointer to the data structure can be replaced by a null pointer). Otherwise, the first two terms are clearly in $\mathcal{O}(\ell\,\lceil\log(r/\ell)\rceil)$ since $\binom{r}{\ell} \leq (re/\ell)^\ell$, whereas the third term can be bounded as follows: if $\ell > \log\log r$, then $\ell \cdot \lceil\log(r/\ell)\rceil \geq \ell > \log\log r$, and if $\ell \leq \log\log r$, then $\ell \cdot \lceil\log(r/\ell)\rceil \geq \log(r/\ell) \geq \log r - \log\log r = \omega(\log\log r)$.

$\mathsf{TreeLCP}(\mathcal{T}_v, P)$ query for $\mathcal{T}_v$ being the subtree of $\mathcal{T}$ rooted at v. If v is the root of $\mathcal{T}$, the query $\mathsf{TreeLCP}_v(\mathcal{T}, P) = \mathsf{TreeLCP}(\mathcal{T}, P)$ is called *rooted*. In computation of $\mathsf{TreeLCP}$, perfect hashing can be used.

Theorem 2.2 ([FKS84]). *A compact trie $\mathcal{T}$ of n static strings can be stored in $\mathcal{O}(n)$ space (in addition to the stored strings themselves) so that a $\mathsf{TreeLCP}_v(\mathcal{T}, P)$ query for node v of $\mathcal{T}$ and a pattern P can be answered in $\mathcal{O}(d)$ time, where d is the string depth of the returned node in $\mathcal{T}_v$.*

Data structures for answering rooted and unrooted $\mathsf{TreeLCP}$ queries on sparse suffix trees were proposed by Cole et al. [CGL04]; the space complexity of unrooted queries was improved by Chan et al. [CLS+10].

Theorem 2.3 ([CLS+10, CGL04]). *Assume we are given compact tries $\mathcal{T}_1, \ldots, \mathcal{T}_t$ of total size N, each containing suffixes of a length-n text T, and that the suffix tree $\widetilde{\mathcal{T}}$ of T is given. There exists a data structure of size $\mathcal{O}(n+N)$ that, given a length-m pattern P, preprocesses P in $\mathcal{O}(m)$ time so that later an unrooted query $\mathsf{TreeLCP}_v(\mathcal{T}_i, P')$, for any tree $\mathcal{T}_i$, $i \in [1\,..\,t]$, its node v, and suffix P' of P, can be answered in $\mathcal{O}(\log\log n)$ time.*

The following lemma is a component of the data structure behind Theorem 2.3.

Lemma 2.4 (Cole et al. [CGL04]). *The suffix tree $\widetilde{\mathcal{T}}$ of a length-n text T can be enhanced with a data structure of size $\mathcal{O}(n)$ that, given a length-m pattern P, returns $\mathsf{TreeLCP}(\widetilde{\mathcal{T}}, U)$ for every suffix U of P in $\mathcal{O}(m)$ total time.*

3 LCPs on Modified Suffixes

The lemmas in this section help us optimize the k-mismatch index for short patterns. We believe that the data structures from this section will find further applications. The proofs of lemmas in this section are deferred to Section 8.

Let $\mathcal{T}$ be a compact trie of a given set of fragments of a length-n text T. We say that $\mathcal{T}$ is given in a *canonical form* if, for every terminal node v of $\mathcal{T}$ that corresponds to a fragment $T[a\,..\,b)$, either $b = n$ or v does not have an outgoing edge along the character $T[b]$ (see Figure 1). Let us note that a sparse suffix tree of T is automatically in a canonical form. Any compact trie $\mathcal{T}$ of fragments of T can be transformed into a tree of the same shape and with the same number of terminals but in a canonical form by repeating the following correcting process: while $\mathcal{T}$ has a terminal v that corresponds to a fragment $T[a\,..\,b)$ and v has an outgoing edge with label $T[b]$, move the terminal v one character down along $T[b]$, effectively replacing the fragment $T[a\,..\,b)$ with $T[a\,..\,b]$.

Unrooted $\mathsf{TreeLCP}$ queries can be answered efficiently also on compact tries of fragments of T given in a canonical form. In short, we compute the result of a $\mathsf{TreeLCP}$ query obtained for the compact tries of the corresponding suffixes of T and trim it, which is possible thanks to the original trees being in a canonical form.

Lemma 3.1. *Theorem 2.3 holds also if $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are compact tries of fragments of T, each given in a canonical form.*

In the proof of the next lemma, we decompose each compact trie of k-modified suffixes into compact tries of fragments of T, each given in a canonical form, and very small tries that correspond to modifications.

Lemma 3.2. *For all constants $k, k' = \mathcal{O}(1)$, Theorem 2.3 holds also if $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are compact tries of ($\leq k$)-modified suffixes of T and queries are for ($\leq k'$)-modified suffixes of P.*

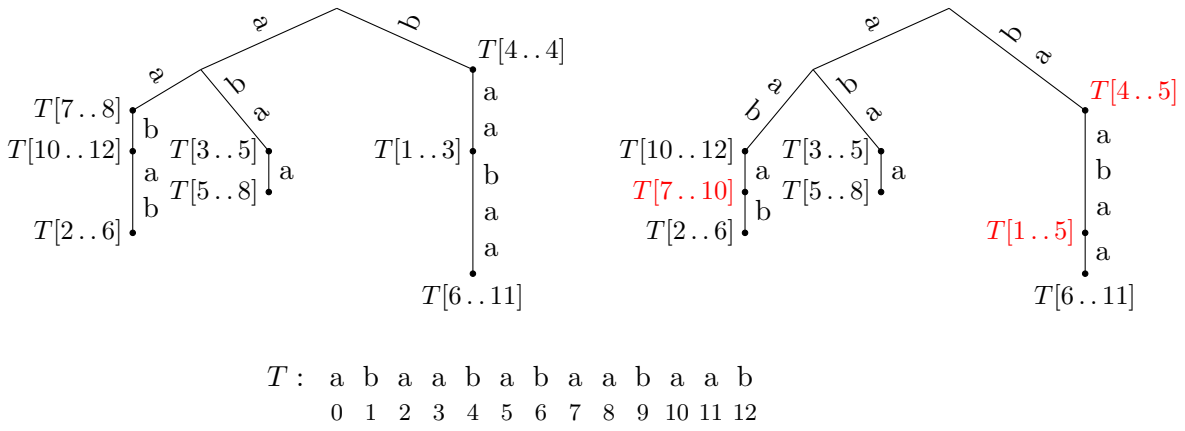

Figure 1: Left: a compact trie of a set of fragments of T. Right: a compact trie of the same shape given in a canonical form; the updated terminals are shown in red.

For superconstant k, k', the space of the data structure in the lemma is $\mathcal{O}(n + Nk)$ and the query time for an unrooted TreeLCP query is $\mathcal{O}((k + k' + 1) \log \log n)$.

The next lemma follows by applying kangaroo jumps [GG87, LV86] with the aid of the data structure of Lemma 2.4.

Lemma 3.3. *A length-n text T can be enhanced with a data structure of size $\mathcal{O}(n)$ that, given a length-m pattern P, preprocesses P in $\mathcal{O}(m)$ time so that later, given a k-modified suffix S of T and a k'-modified suffix P' of P, we can compute $\mathsf{LCP}(S, P')$ in $\mathcal{O}(k + k' + 1)$ time.*

4 k-Mismatch Index for $\sigma = \mathcal{O}(1)$ for Short Patterns

We proceed with an index for small m designed for constant k and σ. We start with a simpler index with worse complexity. Then, in Theorem 1.4, we optimize its complexity.

Fact 4.1. *Let T be a string of length n over an alphabet of size $\sigma = \mathcal{O}(1)$ and $k = \mathcal{O}(1)$. For every $\mu \in [1\,..\,n]$ and $h \in [1\,..\,k)$, there exists a k-mismatch index using $\mathcal{O}(n\mu^h)$ space that can answer queries for patterns of length $m \leq \mu$ in $\mathcal{O}(m^{k-h+1} \log \log n + \mathsf{occ})$ time.*

Proof. Data structure. For every $\kappa \in [1\,..\,h]$ and $j \in [\kappa - 1\,..\,\mu)$, the data structure stores a compact trie $\mathcal{T}_{\kappa,j}$ of κ-modified suffixes S of T such that all substitutions in S are at positions $[0\,..\,j]$ of S and the rightmost (κth) substitution in S is at the jth position of S. Moreover, the data structure contains the suffix tree of T, which we denote as $\mathcal{T}_{0,0}$; for $j > 0$, we assume that $\mathcal{T}_{0,j}$ are empty. For answering rooted TreeLCP queries for compact tries of $(\leq k)$-modified suffixes of T and $(\leq k)$-modified patterns P, we use Lemma 3.2.

For a given $\kappa \in [0\,..\,h]$, there are $\mathcal{O}(n\binom{\mu}{\kappa}\sigma^\kappa) = \mathcal{O}(n\mu^h)$ κ-modified suffixes with modifications at the first μ positions. Each $(\leq h)$-modified suffix belongs to exactly one of the compact tries. Overall, the compact tries use $\mathcal{O}(n\mu^h)$ space (as each edge is represented in $\mathcal{O}(h) = \mathcal{O}(1)$ space). The data structure of Lemma 3.2 uses $\mathcal{O}(n\mu^h)$ total space.

Queries. Assume we are given a query pattern P of length $m \leq \mu$. First, we list h-mismatch occurrences of P. To this end, for each $\kappa \in [0\,..\,h]$ and $j \in (\kappa\,..\,m)$, we look for the locus of P in $\mathcal{T}_{\kappa,j}$ and, if it exists, report all the terminals in the subtree of the computed node. Next, we report k-mismatch occurrences of P that contain at least $h+1$ substitutions. To this end, we generate all

strings P' that are at Hamming distance at least 1 and at most $k-h$ from P. For each such string P', if j is the first position where P and P' mismatch, we look for the locus of P' in each compact trie $\mathcal{T}_{h,i}$ for $i \in [h-1\,..\,j)$ and, if it exists, report all the terminals in the subtree of the computed node. Each k-mismatch occurrence of P is reported exactly once.

The locus of a $(\leq k-h)$-modified pattern in a compact trie can be computed in $\mathcal{O}(\log\log n)$ time using Lemma 3.2. In total, we compute $\mathcal{O}(m)$ times the locus of P to compute h-mismatch occurrences of P and $\mathcal{O}(m^{k-h+1}\sigma^{k-h})$ times the locus of a $(\leq k-h)$-modified P to compute the remaining k-mismatch occurrences of P. The total query time is $\mathcal{O}(m^{k-h+1}\log\log n + \mathsf{occ})$. □

First improvement. We use a folklore construction that covers the interval $[1\,..\,\mu]$ by $\mathcal{O}(\mu)$ (here: just μ) so-called base intervals so that each position in $[1\,..\,\mu]$ is covered by $\mathcal{O}(\log\mu)$ base intervals and each interval $[1\,..\,j]$, for $j \in [1\,..\,\mu]$, can be decomposed into $\mathcal{O}(\log j)$ base intervals. For a positive integer t, by $f(t)$ we denote the maximum integer power of two that divides t. With the formula $f(t) = \frac{1}{2}((t \text{ xor } (t-1)) + 1)$, values $f(t)$ can be computed in $\mathcal{O}(1)$ time in the word RAM. We use base intervals $[t-f(t)\,..\,t)$ for $t \in [1\,..\,\mu]$. For a positive integer j, its *f-sequence* is defined as a decreasing sequence of positive integers that is constructed as follows. We start with element j. If the last element was t, then the next element is $t-f(t)$, unless $t-f(t)=0$, in which case the sequence is terminated. Then the set of right endpoints of base intervals that cover the interval $[1\,..\,j]$ is exactly the f-sequence of j.

The improved data structure stores, for every $\kappa \in [0\,..\,h]$ and $j \in [1\,..\,\mu]$, a compact trie $\mathcal{T}'_{\kappa,j}$ that consists of modified suffixes from $\mathcal{T}_{\kappa,i}$ for all $i \in [j-f(j)\,..\,j)$. This way the modified suffixes from compact trie $\mathcal{T}_{\kappa,i}$ are stored in $\mathcal{O}(\log\mu)$ compact tries $\mathcal{T}'_{\kappa,j}$. Thus, the space complexity grows just by a factor of $\mathcal{O}(\log\mu)$. Naturally, we construct data structures for unrooted $\mathsf{TreeLCP}$ queries in tries $\mathcal{T}'_{\kappa,j}$. The total space complexity is $\mathcal{O}(n\mu^h\log\mu)$.

The query algorithm is adjusted as follows. First, h-mismatch occurrences of P are reported by computing the locus of P in $\mathcal{T}'_{\kappa,t}$ for all $\kappa \in [0\,..\,h]$ and t in the f-sequence of m. Then, to compute k-mismatch occurrences with at least $h+1$ substitutions, we generate all modified patterns P' that are at Hamming distance at least 1 and at most $k-h$ from P. For each such string P', if j is the first position where P and P' mismatch, we find the locus of P' in each compact trie $\mathcal{T}'_{h,t}$ for t in the f-sequence of j. Each k-mismatch occurrence of P is still reported exactly once.

The length of an f-sequence of $j \in [1\,..\,m]$ is $\mathcal{O}(\log m)$. We ask $\mathcal{O}(k\log m)$ $\mathsf{TreeLCP}$ queries in the compact tries to compute h-mismatch occurrences of P and $\mathcal{O}(m^{k-h}\log m)$ $\mathsf{TreeLCP}$ queries to compute the remaining k-mismatch occurrences. Thus, the time complexity of a query is $\mathcal{O}(m^{k-h}\log m\log\log n + \mathsf{occ})$.

Second improvement. In the lemma below, we extend rank and select queries of Raman et al. [RRR07] to storing a collection of increasing sequences. The proof applies Jensen's inequality.

Lemma 4.2. *A collection of c increasing sequences of total length ℓ, each composed of integers in $[1\,..\,r]$, can be stored using $\mathcal{O}(\ell(1+\log(cr/\ell)))$ bits to support rank and select queries on each sequence in $\mathcal{O}(1)$ time.*

Proof. Let $\ell_1, \ell_2, \ldots, \ell_c$ be the lengths of the subsequent sequences. By Theorem 2.1, the total number of bits required to represent all sequences is $\mathcal{O}(c + \sum_{i=1}^{c} \ell_i(1+\log(r/\ell_i)))$. Since $f(x) = x\log(r/x)$ is concave, this value is maximized for $\ell_i = \ell/c$; then, the total size is as required (as $c \leq \ell$). □

Let us fix a tree $\mathcal{T}'_{\kappa,j}$. Let $S_1, S_2, \ldots, S_t$ ($S_1 < S_2 < \cdots < S_t$) be strings corresponding to terminals of $\mathcal{T}'_{\kappa,j}$. We select every $\lfloor\log n\rfloor$th terminal from the sequence as well as S_1 and S_t and

form a compact trie $\mathcal{T}''_{\kappa,j}$ of the strings corresponding to the selected terminals. The terminal S_i in $\mathcal{T}''_{\kappa,j}$ is numbered by i. The new tries take $\mathcal{O}(\mu + n\mu^h \log \mu / \log n)$ space in total. As before, after the improvement we construct data structures for unrooted TreeLCP queries in tries $\mathcal{T}''_{\kappa,j}$. The trie $\mathcal{T}'_{\kappa,j}$ is not stored; instead, we design a compact data structure that allows us to retrieve S_i in $\mathcal{O}(1)$ time using Lemma 4.2, as shown in the lemma below.

Lemma 4.3. *If $\mu = \Omega(\log n)$, there is a data structure of size $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$ that, given $\kappa \in [0\,..\,h]$, $j \in [1\,..\,\mu]$, and $i \in [1\,..\,|\mathcal{T}'_{\kappa,j}|]$, returns the i-th lexicographically terminal of $\mathcal{T}'_{\kappa,j}$ in $\mathcal{O}(1)$ time.*

Proof. Data structure. We store a lexicographically sorted list $\mathcal{L}$ of all $(\le h-1)$-modified suffixes of T with modifications at the first μ positions only. The subsequent elements of $\mathcal{L}$ are denoted as $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{|\mathcal{L}|}$. Each modified suffix can be represented in $\mathcal{O}(h) = \mathcal{O}(1)$ space and $|\mathcal{L}| = \mathcal{O}(n\mu^{h-1})$, so $\mathcal{O}(n\mu^{h-1}) \subseteq \mathcal{O}(n\mu^h / \log n)$ space is required.

For each terminal $(\le h)$-modified suffix S in $\mathcal{T}'_{\kappa,j}$, a triad (p, c_{old}, c_{new}) is stored such that the rightmost modification in S is at position p, the character before modification is c_{old}, and the character after modification is c_{new}. The sequence of triads for all terminals of $\mathcal{T}'_{\kappa,j}$ in lexicographic order is denoted as $\mathit{Triads}_{\kappa,j}$. If S contains no modifications, we can set $p = -1$. Each triad is represented in $\mathcal{O}(\log \mu)$ bits and the number of triads is the same as the total size of tries $\mathcal{T}'_{\kappa,j}$, that is, $\mathcal{O}(n\mu^h \log \mu)$. Hence, the space complexity of this component $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$.

For each trie $\mathcal{T}'_{\kappa,j}$ and triad (p, c_{old}, c_{new}), we store an increasing sequence $\mathit{MyTriad}_{\kappa,j,p,c_{old},c_{new}}$ of integers in $[1\,..\,|\mathcal{T}'_{\kappa,j}|]$ such that x occurs in this sequence if and only if the lexicographically xth terminal of $\mathcal{T}'_{\kappa,j}$ has a triad (p, c_{old}, c_{new}) (i.e., $\mathit{Triads}_{\kappa,j}[x] = (p, c_{old}, c_{new})$). Thus, for a given trie $\mathcal{T}'_{\kappa,j}$, we form $\mathcal{O}(\mu)$ increasing sequences of total length $|\mathcal{T}'_{\kappa,j}|$ and values in $[1\,..\,|\mathcal{T}'_{\kappa,j}|]$ and store them using Lemma 4.2. This part of the data structure takes $\mathcal{O}(|\mathcal{T}'_{\kappa,j}| \log \mu / \log n)$ space per each trie, for a total of $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$ space.

Finally, for each trie $\mathcal{T}'_{\kappa,j}$ and valid triad (p, c_{old}, c_{new}), we store a sequence $\mathit{Cut}_{\kappa,j,p,c_{old},c_{new}}$ of integers in $[1\,..\,|\mathcal{L}|]$ such that x occurs as the ith element in this sequence if and only if the lexicographically ith terminal with triad (p, c_{old}, c_{new}) in $\mathcal{T}'_{\kappa,j}$ (i.e., the globally $\mathit{MyTriad}_{\kappa,j,p,c_{old},c_{new}}[i]$'th terminal in the trie) without its rightmost modification (i.e., the one at position p) equals $\mathcal{L}_x$. The key observation is that the sequence $\mathit{Cut}_{\kappa,j,p,c_{old},c_{new}}$ is *increasing* because all the considered terminals have the same character at position p (c_{new}) and all the terminals without the last modification also have the same character at this position (c_{old}). Here $\mathcal{O}(\mu^2)$ increasing sequences are stored using Lemma 4.2. Each terminal of each trie implies exactly one element in one of the sequences, so the total length of the sequences is $\mathcal{O}(n\mu^k \log \mu)$ and their values are in $[1\,..\,|\mathcal{L}|]$; thus again $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$ space is used.

Queries. Say we need to compute a terminal specified by indices κ, j, i. First we recover the triad $\mathit{Triads}_{\kappa,j}[i] = (p, c_{old}, c_{new})$. With a rank query in sequence $\mathit{MyTriad}_{\kappa,j,p,c_{old},c_{new}}$, we recover an index a of this sequence that stores element i. With a select query on sequence $\mathit{Cut}_{\kappa,j,p,c_{old},c_{new}}$, we compute the ath element of this sequence, say x. In the end, we return the $(\le h-1)$-modified string $\mathcal{L}_x$ with an additional substitution $\mathcal{L}_x[p] := c_{new}$. Each step takes $\mathcal{O}(1)$ time by Lemma 4.2. □

Remark 4.4. Without the assumption that $\mu = \Omega(\log n)$, the space complexity in Lemma 4.3 becomes $\mathcal{O}(n\mu^{h-1} + n\mu^h / \log n)$.

The next simple technical lemma is also reused in our general compact index. We use static predecessor/successor data structure. Such a data structure is constructed for a specified set A of integers. A predecessor query to A, given an integer x, returns $\max\{y \in A : y \le x\}$. A successor query returns $\min\{y \in A : y \ge x\}$. Using y-fast tries [Wil83] one can obtain a static $\mathcal{O}(m)$-size

data structure on a set A of m sorted keys from $[1\,..\,n]$ that answers predecessor/successor queries in A in $\mathcal{O}(\log\log n)$ time.

Lemma 4.5. *Let $k = \mathcal{O}(1)$ and $\mathcal{T}_1, \ldots, \mathcal{T}_t$ be compact tries of $(\leq k)$-modified suffixes of T. Further, let $\mathcal{T}'_i$, for $i \in [1\,..\,t]$, be a compact trie formed by selecting every $\lfloor \log n \rfloor$th terminal of $\mathcal{T}_i$ in the lexicographic order as well as the first and the last terminal. We assume that tries $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are not stored; instead, we can compute in $\mathcal{O}(1)$ time the lexicographically jth terminal of $\mathcal{T}_i$.*

Let $k' = \mathcal{O}(1)$ and P be a string pattern such that for every $(\leq k')$-modified suffix P' of P, any query $\mathsf{TreeLCP}(\mathcal{T}'_i, P')$ can be answered in $\mathcal{O}(\log\log n)$ time and any $(\leq k)$-modified suffix of T can be lexicographically compared with P' in $\mathcal{O}(1)$ time. Using $\mathcal{O}(\sum_{i=1}^{t} |\mathcal{T}'_i|)$ additional space, for a given index $i \in [1\,..\,t]$ and $(\leq k')$-modified suffix P' of P, we can compute the range of terminals in $\mathcal{T}_i$ that have P' as a prefix in $\mathcal{O}(\log\log n)$ time.

Proof. Assume that terminals in $\mathcal{T}'_i$ are numbered the same as the original terminals from $\mathcal{T}_i$. Whenever the locus of a $(\leq k')$-modified suffix P' of P in $\mathcal{T}_i$ is to be computed, a $\mathsf{TreeLCP}(\mathcal{T}'_i, P')$ query is issued instead. Let v be the returned node. The remainder of the algorithm depends on if the string depth of v is $|P'|$. If so, let x_{min} and x_{max} be the minimum and maximum number of a terminal in the subtree of v in $\mathcal{T}'_i$. The desired sublist of numbers of terminals of $\mathcal{T}_i$ contains the interval $[x_{min}\,..\,x_{max}]$. Let y_{max} and z_{min} be the maximum number of a terminal in $\mathcal{T}'_i$ that is smaller than x_{min} and the minimum number of a terminal that is greater than x_{max}, respectively. If y_{max} exists, we need to further inspect terminals of $\mathcal{T}_i$ with numbers in $(y_{max}\,..\,x_{min})$. Similarly, if z_{min} exists, we need to further inspect terminals of $\mathcal{T}_i$ with numbers in $(x_{max}\,..\,z_{min})$.

Now assume that the string depth d of v is smaller than $|P'|$. Let x_{left} be the maximum number of a terminal in $\mathcal{T}'_i$ such that the represented modified suffix S satisfies $S < P'[0\,..\,d]$. If x_{left} does not exist, we are done. Otherwise, let x_{right} be the minimum number of a terminal in $\mathcal{T}'_i$ that is greater than x_{min}. Then we need to inspect terminals of $\mathcal{T}'_i$ with numbers in $(x_{left}\,..\,x_{right})$.

Claim 4.6. In each case, the required terminal numbers can be retrieved from $\mathcal{T}'_i$ in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(|\mathcal{T}'_i|)$ data stored for $\mathcal{T}'_i$.

Proof. We store the following data for $\mathcal{T}'_i$: (1) for each explicit node v, the minimal and maximal number of terminal in v's subtree; (2) for each terminal number in $\mathcal{T}'_i$, its predecessor and successor terminal number; (3) for each explicit node v, the maximum terminal with label that does not exceed the path label of v; (4) for each explicit node v, a predecessor data structure of [Wil83] on children of v indexed by the first characters of their edges.

If the string depth of v is $|P'|$, we compute x_{min} and x_{max} using (1) and y_{max}, z_{min} using (2). Otherwise, x_{left} is computed as follows. Using (4), we compute the child of v along the maximum character smaller than $P'[d]$; if such a child exists, we report the maximal terminal in its subtree using (1). If it does not exist, we choose the terminal according to (3). The predecessor query takes $\mathcal{O}(\log\log n)$ time and the remaining queries take $\mathcal{O}(1)$ time. The space of the data structures (1)–(4) is $\mathcal{O}(|\mathcal{T}'_i|)$. □

In both cases, after $\mathcal{O}(\log\log n)$ time, we are left with $\mathcal{O}(1)$ intervals of terminal numbers of length $\mathcal{O}(\log n)$ to be inspected. We can retrieve the jth terminal of $\mathcal{T}_i$ in $\mathcal{O}(1)$ time and compare it lexicographically with P' in $\mathcal{O}(1)$ time. Thus, we can use binary search to find a subinterval of a given interval of terminals that corresponds to terminals whose prefix is P'. This requires just $\mathcal{O}(\log\log n)$ time. □

We apply Lemma 4.5 to tries $\mathcal{T}'_{\kappa,j}$ and tries after sampling $\mathcal{T}''_{\kappa,j}$. By Lemma 4.3, we can compute the jth terminal of $\mathcal{T}'_{\kappa,j}$ in $\mathcal{O}(1)$ time using $\mathcal{O}(n\mu^h \log^2 \mu / \log n)$ space. By Lemma 3.3, we can

compare any ($\leq k'$)-modified suffix P' of P with any ($\leq k$)-modified suffix S of T in $\mathcal{O}(1)$ time using just $\mathcal{O}(n)$ space. With Lemma 4.5, we can compute an interval of terminals of $\mathcal{T}'_{\kappa,j}$ that have P' as a prefix. This gives a total of $\mathcal{O}(m^{k-h}\log m\log\log n)$ time. Then all occurrences from the interval can be reported in $\mathcal{O}(1)$ time per occurrence, using Lemma 4.3.

With both improvements, we obtain Theorem 1.4. This, in turn, yields the trade-off data structure.

Theorem 1.2. *For every text of length n over an alphabet of size $\sigma = \mathcal{O}(1)$, integer threshold $k = \mathcal{O}(1)$, and real constant $\varepsilon > 0$, there exists a k-mismatch index of size $\mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+2}} n)$ for even k or $\mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+1}} n)$ for odd k that answers queries in $\mathcal{O}(\log^k n\log\log n + m + \mathsf{occ})$ time, where m is the length of the pattern and occ is the number of k-mismatch occurrences.*

Proof. Recall that the index of Theorem 1.4 takes $\mathcal{O}(n\mu^h\log^2\mu/\log n)$ space and answers queries for patterns of length $m \leq \mu$ in $\mathcal{O}(m^{k-h}\log m\log\log n + \mathsf{occ})$ time. The index of Theorem 1.5 takes $\mathcal{O}(n + n\log^{k+\varepsilon} n/\gamma)$ space and answers queries for patterns of length $m \geq (k+1)\gamma$ in $\mathcal{O}(\log^k n\log\log n + m + \mathsf{occ})$ time.

Assume first that k is even. We use the data structure of Theorem 1.4 for $\mu = \left\lfloor \log^{\frac{2k+2}{k+2}} n \right\rfloor$ with $h = k/2$ and the data structure for long patterns (Theorem 1.5) for $\gamma = \mu/(k+1)$. The former uses

$$\mathcal{O}(n\log^{\frac{k(k+1)}{k+2}-1} n\cdot(\log\log n)^2) = \mathcal{O}(n\log^{k-2+\frac{2}{k+2}} n\cdot(\log\log n)^2) \subset \mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+2}} n)$$

space and answers queries in $\mathcal{O}(\log^{k-1+\frac{2}{k+2}} n(\log\log n)^2 + \mathsf{occ}) \subset \mathcal{O}(\log^k n + \mathsf{occ})$ time. The latter uses $\mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+2}} n)$ space and answers queries in $\mathcal{O}(\log^k n\log\log n + m + \mathsf{occ})$ time. The bounds for even k follow.

For odd k, we use the data structure of Theorem 1.4 for $\mu = \left\lfloor \log^{\frac{2k}{k+1}-\varepsilon'} n \right\rfloor$ with $\varepsilon' = \frac{2}{k+1}\varepsilon$ and $h = (k-1)/2$ and the data structure for long patterns (Theorem 1.5) for $\gamma = \mu/(k+1)$. The former uses

$$\mathcal{O}(n\log^{\frac{k(k-1)}{k+1}-1-\frac{k-1}{2}\varepsilon'} n(\log\log n)^2) \subset \mathcal{O}(n\log^{k-3+\varepsilon+\frac{2}{k+1}} n(\log\log n)^2) \subset \mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+1}} n)$$

space and answers queries in $\mathcal{O}(\log^{k-\frac{k+1}{2}\varepsilon'} n(\log\log n)^2 + \mathsf{occ}) = \mathcal{O}(\log^{k-\varepsilon} n + \mathsf{occ})$ time. The latter uses $\mathcal{O}(n\log^{k-2+\varepsilon'+\frac{2}{k+1}} n) \subset \mathcal{O}(n\log^{k-2+\varepsilon+\frac{2}{k+1}} n)$ space; query time is $\mathcal{O}(\log^k n\log\log n + m + \mathsf{occ})$. The bounds for odd k follow. □

5 Errata Tree for Hamming Distance

The data structure for k-mismatch indexing of Cole et al. [CGL04] actually solves an "all-to-all" problem, in which we are given x suffixes $R_1, R_2, \ldots, R_x$ of the text T, and for a query with pattern P, given a suffix P' of P we are to report all $i \in [1\,..\,x]$ such that P' matches a prefix of R_i with up to k substitutions. It is assumed that the suffix tree of the text T is accessible. The next theorem summarizes the complexity of the data structure.

Theorem 5.1 ([CGL04])**.** *Let T be a text of length n. The k-errata tree for Hamming distance for x suffixes $R_1, \ldots, R_x$ of T uses space $\mathcal{O}(n+x\log^k x)$ and, for a pattern P of length m, after $\mathcal{O}(m)$ time preprocessing, for any suffix P' of P can compute the set $\{i \in [1\,..\,x] : \delta_H(P', R_i[0\,..\,|P'|)) \leq k\}$ in $\mathcal{O}(\log^k x\log\log x + \mathsf{occ})$ time, where occ is the size of the set.*

Let us list basic properties of the k-errata tree. The k-errata tree is a collection of compact tries. Each compact trie is assigned a level in the data structure. The compact trie at level 0 is the compact trie of the suffixes $R_1, \ldots, R_x$. At each level $i \in [1\,..\,k]$, each compact trie is of one of the following two types:

(1) a sparse suffix tree of T attached to a root with a wildcard-character edge; or

(2) a rooted path π with several sparse suffix trees of T attached at different locations of π.

The rooted path π is called the *main path* of the compact trie. Each main path has a string label being a suffix of T. For each sparse suffix tree in the k-errata tree (in particular, for each main path), an unrooted TreeLCP data structure is stored (Theorem 2.3).

A $(k-1)$-errata tree is simply composed of compact tries at levels $0, 1, \ldots, k-1$ of a k-errata tree.

Let us describe in more detail how compact tries at subsequent levels of the k-errata tree are formed. Let $\mathcal{T}$ be a compact trie at level $i \in [0\,..\,k)$. We use a heavy-light decomposition of $\mathcal{T}$, that is, we classify explicit nodes of $\mathcal{T}$ as light and heavy. The root is always light. A non-root node is called heavy if the size of its subtree (measured as the number of terminals in the subtree) is maximal among all the subtrees of its siblings (breaking ties arbitrarily); otherwise the node is light. A heavy path is a path that starts at a light node and goes down through heavy nodes until a heavy leaf is reached.

Let π be a heavy path of $\mathcal{T}$. A subtree of $\mathcal{T}$ rooted at an explicit child u of an explicit node v from π, together with the compact edge from v to u, is called an *off-path subtree* if u is not on π. The off-path subtree such that the first character on the edge from v to u is a is denoted as $\mathcal{T}_{v,a}$.

The following types of *substitution trees* are formed from off-path subtrees:

(a) $\mathcal{T}^s_{v,a}$ which is $\mathcal{T}_{v,a}$ with the first character substituted from a to a special character ψ

(b) $\mathcal{T}^s_v$ which is the merge of all substitution trees $\mathcal{T}^s_{v,a}$, over all characters a different from the next character b after v on the heavy path π, with character ψ replaced by character b.

Here, a merge of subtrees is defined naturally as a compact trie representing all terminals from subsequent subtrees.

Finally, substitution trees are organized into group trees. Specifically, substitution trees of type (a) are merged into group trees for each node v separately, and substitution trees of type (b) are merged over a whole heavy path. For a collection of substitution trees, a group tree being the merge of all substitution trees is formed, and then it is subdivided into smaller group trees consisting of subsets of substitution trees. The order of merging is lexicographic by character a in type (a) and by the string depth of node v in type (b). Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$ be all substitution trees being merged into group trees, for a given node v in case (a) or for a given path π in case (b). Then the group trees are organized as in weighted search trees so that for a given substitution tree $\mathcal{T}_j$, the sizes (i.e., numbers of terminals) of group trees that contain $\mathcal{T}_j$ decrease top to bottom, and halve at least in every second step. The group trees become compact tries at level $i+1$ of the data structure.

Technically, for substitution trees of type (b), in order to make the merging meaningful, each substitution tree is assumed to contain the path from v to the root of the heavy path π. Then the substitution tree (hence, the resulting group tree) is actually a compact trie of 1-modified suffixes of T. The modification takes place only on the heavy path π, so the trees hanging from the main path in the group tree are sparse suffix trees of T. We treat the main path as heavy in the next step of recursion.

Let $\mathcal{T}$ be a compact trie in the k-errata tree. Each terminal in $\mathcal{T}$ can be assigned a *label* ℓ being a position in $[0\,..\,n)$. (Actually, it is a position in $\{n-|R_j| \,:\, j\in[1\,..\,x]\}$.) If a node v in $\mathcal{T}$ is computed as a result of a query, then for every terminal in the subtree of v, the query algorithm reports a k-mismatch occurrence at position equal to the terminal's label.

We prove the following lemma using the recursive construction of k-errata trees.

Lemma 5.2. *In a k-errata tree built from x suffixes of T for $k=\mathcal{O}(1)$, each terminal label occurs in compact tries at most $\mathcal{O}(\log^k x)$ times overall.*

Proof. Let ℓ be a terminal label in the compact trie at level 0. If $\mathcal{T}$ is a compact trie at level i and contains a terminal with label ℓ, then by construction every group tree formed from $\mathcal{T}$ contains at most one terminal with label ℓ. By induction, every terminal label is present in every compact trie of a k-errata tree at most once.

Let $\mathcal{T}$ be a compact trie at level $i\in[0\,..\,k)$ in the k-errata tree that contains a terminal with label ℓ. For a heavy path C in $\mathcal{T}$, by $\mathcal{T}_C$ we denote the subtree of $\mathcal{T}$ rooted at the root of C. Let $C_1,C_2,\ldots,C_t$ be all heavy paths, top-down, visited on the path from the root of $\mathcal{T}$ to the terminal with label ℓ. The organization of group trees implies that, for every $j\in[1\,..\,t)$, the terminal label ℓ occurs in $\mathcal{O}(1+\log|\mathcal{T}_{C_j}|-\log|\mathcal{T}_{C_{j+1}}|)$ group trees (of both types) formed from heavy path C_j; for $j=t$, in $\mathcal{O}(\log|\mathcal{T}_{C_t}|)$ group trees. This telescopes to $\mathcal{O}(\log|\mathcal{T}_{C_1}|)=\mathcal{O}(\log|\mathcal{T}|)=\mathcal{O}(\log x)$ group trees, i.e., $\mathcal{O}(\log x)$ compact tries at level $i+1$ formed from $\mathcal{T}$ that contain a terminal with label ℓ. Overall, terminal label ℓ is present in $\mathcal{O}(\sum_{i=0}^{k}\log^i x)=\mathcal{O}(\log^k x)$ compact tries. □

A k-mismatch query is asked for a pattern P for the compact trie at level 0 and with a budget of k mismatches and is handled recursively, level by level. A query in a compact trie $\mathcal{T}$ at level i for a suffix P' of P with budget k' reduces to possibly several queries for suffixes of P' in compact tries at level $i+1$ and budget at most $k'-1$. In case a k-mismatch occurrence of P in T is actually a k'-mismatch occurrence for some $k'<k$, it is reported in a compact trie at level $\leq k'$. In this case, a node v in the compact trie is computed and all terminals in the subtree of v are reported. In total, $\mathcal{O}(\log^k x)$ such nodes are returned in a query of Theorem 5.1.

When the query algorithm is called recursively for a compact trie $\mathcal{T}$, a suffix P' of the pattern P, and budget k', its goal is to report labels of all terminals in $\mathcal{T}$ whose length-$|P'|$ prefix is at Hamming distance at most k' from P'. Some of these terminals are located directly by traversing $\mathcal{T}$; others are identified in recursive calls.

6 k-Mismatch Index for Long Patterns

For a string U, an integer $p\in[1\,..\,|U|]$ is called a *period* of U if $U[i]=U[i+p]$ holds for all $i\in[0\,..\,|U|-p)$. The smallest period of U is denoted as $\mathsf{per}(U)$.

We use the notion of string synchronizing sets of Kempa and Kociumaka [KK19]. For a string T of length n and a positive integer $\tau\leq n/2$, a set $A\subseteq[1\,..\,n-2\tau+1]$ is a *τ-synchronizing set* of T if it satisfies the following two conditions:

1. Consistency: If $T[i\,..\,i+2\tau)$ matches $T[j\,..\,j+2\tau)$, then $i\in A$ if and only if $j\in A$.
2. Density: For $i\in[1\,..\,n-3\tau+2]$, $A\cap[i\,..\,i+\tau)=\emptyset$ if and only if $\mathsf{per}(T[i\,..\,i+3\tau-1))\leq\tau/3$.

Let us fix a τ-synchronizing set A. If a position i belongs to A, we call $T[i\,..\,i+2\tau)$ a *τ-synchronizing fragment* of T.

Theorem 6.1 ([KK19, Proposition 8.10])**.** *For a string T of length n and $\tau\leq n/2$, there exists a τ-synchronizing set of T of size $\mathcal{O}(n/\tau)$.*

We note that the synchronizing set can be constructed in $\mathcal{O}(n)$ time if T is over an integer alphabet.

The case that a long substring of a string does not contain synchronizing positions is covered by considering periodic substrings of T called *runs*. String U is called *periodic* if $2 \cdot \mathsf{per}(U) \leq |U|$. A *run* in T is a periodic substring $T[i\,..\,j]$ that is maximal, i.e. such that $T[i-1] \neq T[i-1+p]$ and $T[j+1] \neq T[j+1-p]$ for $p = \mathsf{per}(T[i\,..\,j])$ provided that the respective positions exist. All runs in a string can be computed in linear time [BII+17, EF21, KK99]. A run R is called a *τ-run* if $|R| \geq 3\tau - 1$ and $\mathsf{per}(R) \leq \tau/3$. We use the following lemma.

Lemma 6.2 ([KK19, Section 6.1.2], [CKPR21, Lemma 10])**.** *A string of length n contains $\mathcal{O}(n/\tau)$ τ-runs, for $\tau \in [1\,..\,n]$.*

We also use the notion of *misperiods* that originates from Charalampopoulos et al. [CKP+21]. We say that position a in string S is a *misperiod* with respect to a substring $S[i\,..\,j)$ if $S[a] \neq S[b]$ where b is the unique position such that $b \in [i\,..\,j)$ and $(j-i)$ divides $(b-a)$. We define the set $\mathsf{LeftMisper}_k(S,i,j)$ as the set of k maximal misperiods that are smaller than i and $\mathsf{RightMisper}_k(S,i,j)$ as the set of k minimal misperiods that are at least j. Each of the sets can have less than k elements if the corresponding misperiods do not exist. We also define $\mathsf{Misper}_k(S,i,j) = \mathsf{LeftMisper}_k(S,i,j) \cup \mathsf{RightMisper}_k(S,i,j)$ and $\mathsf{Misper}(S,i,j) = \mathsf{Misper}_{|S|}(S,i,j)$.

Lemma 6.3 ([CKP+21, Lemma 13])**.** *Assume that $|U| = |V|$, $\delta_H(U,V) \leq k$, and that $U[i\,..\,j) = V[i\,..\,j)$. Let $I = \mathsf{Misper}_{k+1}(U,i,j)$ and $I' = \mathsf{Misper}_{k+1}(V,i,j)$. If $I \cap I' = \emptyset$, then $\delta_H(U,V) = |I| + |I'|$, $I = \mathsf{Misper}(U,i,j)$, and $I' = \mathsf{Misper}(V,i,j)$.*

We design a data structure that answers k-mismatch pattern queries with patterns of length $m \geq (k+1)\gamma$, for a specified integer $\gamma \geq 2$. We select two sets of positions in T, A_1 and A_2, called *anchors*. The set A_1 is a τ-synchronizing set of T for $\tau = \lfloor \gamma/3 \rfloor$. The set A_2 is a union of sets $\mathsf{Misper}_{k+1}(T,i,i+p)$ over all τ-runs $T[i\,..\,j]$ in T where $p = \mathsf{per}(T[i\,..\,j])$.

Lemma 6.4. *Let T be a text of length n and $\gamma \in [2\,..\,\lfloor n/(k+1) \rfloor]$. Then the sets of anchors A_1, A_2 for T satisfy $|A_1| = \mathcal{O}(n/\gamma)$ and $|A_2| = \mathcal{O}(nk/\gamma)$.*

Proof. By Theorem 6.1, $|A_1| = \mathcal{O}(n/\tau)$. By Lemma 6.2, $|A_2| = \mathcal{O}(nk/\tau)$. Finally, $\tau = \Theta(\gamma)$. □

Given a query for a pattern P of length $m \geq (k+1)\gamma$, we select two sets of positions in P, B_1 and B_2, also called *anchors*. We partition a prefix of P into $k+1$ substrings of length γ. For each $i \in [0\,..\,k]$, if $P[i\gamma\,..\,(i+1)\gamma)$ is a substring of the text T, the set B_1 contains the starting position j of the leftmost 2τ-length string that occurs in $P[i\gamma\,..\,(i+1)\gamma)$ and is a τ-synchronizing fragment of T, provided that such j exists. That is, $j = \min\{j' \in [i\gamma\,..\,(i+1)\gamma - 2\tau) \,:\, P[j'\,..\,j'+2\tau) = T[a\,..\,a+2\tau)$ for some $a \in A_1\}$, provided that the set under min is non-empty.

For every $i \in [0\,..\,k]$, if $P[i\gamma\,..\,(i+1)\gamma)$ is a substring of the text T and $p := \mathsf{per}(P[i\gamma\,..\,(i+1)\gamma)) \leq \tau/3$, the misperiods $\mathsf{Misper}_{k+1}(P, i\gamma, i\gamma+p)$ are inserted to B_2.

Lemma 6.5. *Let T be a text of length n and $\gamma \in [1\,..\,\lfloor n/(k+1) \rfloor]$. There is a data structure of size $\mathcal{O}(n)$ using which, given a pattern P of length $m \geq (k+1)\gamma$, one can construct its sets of anchors B_1, B_2 in $\mathcal{O}(mk)$ time. Moreover, we have $|B_1| \leq k+1$ and $|B_2| \leq 2(k+1)^2$.*

Proof. As for the sizes of sets, we have $|B_1| \leq k+1$ as in each $P[i\gamma\,..\,(i+1)\gamma)$, for $i \in [0\,..\,k]$, we select at most one position. Moreover, $|B_2| \leq 2(k+1)^2$ as for each $P[i\gamma\,..\,(i+1)\gamma)$ that is highly periodic, at most $k+1$ left misperiods and at most $k+1$ right misperiods are inserted to B_2.

Data structure: We store the suffix tree of T enhanced with the data structure for $\mathsf{TreeLCP}$ computation of Theorem 2.2. Each node of the suffix tree stores the label of some terminal node

in its subtree. We also use the data structure of Lemma 2.4. Finally, for each position $i \in [0\,..\,n)$, we store the smallest anchor in $A_1 \cap [i\,..\,n)$, if it exists. The data structures use $\mathcal{O}(n)$ space.

Queries: For each $i \in [0\,..\,k]$, we check if $P[i\gamma\,..\,(i+1)\gamma)$ has a locus in the suffix tree of T. This takes $\mathcal{O}(m)$ time in total by Theorem 2.2. We compute anchors based on values of i for which the locus exists.

To compute the set B_1, if the locus is v, let ℓ be the label of any terminal node in the subtree of v. We set $a = \min(A_1 \cap [\ell\,..\,n))$. If a exists and $a - \ell < \gamma - 2\tau$, $a - \ell + i\gamma$ is inserted to B_1. The algorithm uses just $\mathcal{O}(k)$ time in total. Correctness of the query algorithm follows by the consistency condition of a synchronizing set.

To compute the set B_2 for a given $i \in [0\,..\,k]$, we compute the smallest period p of $P[i\gamma\,..\,(i+1)\gamma)$ using the Morris-Pratt algorithm [KJP77] in $\mathcal{O}(\gamma)$ time, for a total of $\mathcal{O}(m)$ time. If $p \leq \tau/3$, we compute the set $\mathsf{Misper}_{k+1}(P, i\gamma, i\gamma + p)$ by definition in $\mathcal{O}(m)$ time and insert its elements to B_2. We spend $\mathcal{O}(m)$ time for a given i, which totals in $\mathcal{O}(km)$ time. □

Definition 6.6. We say that pattern P has a *k-nearly periodic occurrence* $T' = T[j\,..\,j+m)$ in text T if $\delta_H(P, T') \leq k$ and there are integers $p \in [1\,..\,\lfloor \tau/3 \rfloor]$ and $i \in [0\,..\,k]$ such that $\mathsf{per}(P[i\gamma\,..\,(i+1)\gamma)) = p$, $P[i\gamma\,..\,(i+1)\gamma) = T'[i\gamma\,..\,(i+1)\gamma)$ and $\delta_H(P, T') = |\mathsf{Misper}(P, i\gamma, i\gamma + p)| + |\mathsf{Misper}(T', i\gamma, i\gamma + p)|$.

For two sets A, B of integers, we denote $A \oplus B = \{a + b : a \in A, b \in B\}$ and $A \ominus B = \{a - b : a \in A, b \in B\}$. For an integer j, we denote $A \oplus j = A \oplus \{j\}$, $A \ominus j = A \ominus \{j\}$.

Lemma 6.7. *Let T be a text of length n, $\gamma \in [1\,..\,\lfloor n/(k+1) \rfloor]$, and P be a pattern of length $m \geq (k+1)\gamma$. Every k-mismatch occurrence of P in T starts at a position in $(A_1 \ominus B_1) \cup (A_2 \ominus B_2)$ or is a k-nearly periodic occurrence.*

Proof. Let $j \in [0\,..\,n-m]$ satisfy $\delta_H(T[j\,..\,j+m), P) \leq k$. We will show that $j \in (A_1 \ominus B_1) \cup (A_2 \ominus B_2)$ or $T' = T[j\,..\,j+m)$ is a k-nearly periodic occurrence.

As $m \geq (k+1)\gamma$, at least one of the fragments $P[i\gamma\,..\,(i+1)\gamma)$, for $i \in [0\,..\,k]$, matches the corresponding fragment $T[j+i\gamma\,..\,j+(i+1)\gamma)$ exactly. Let it be the fragment $P_0 = P[i_0\gamma\,..\,(i_0+1)\gamma)$.

If $\mathsf{per}(P_0) > \tau/3$, by the density condition of synchronizers, A_1 contains an element in $[j + i_0\gamma\,..\,j + i_0\gamma + \tau)$. Let $a \in A_1$ be the minimum such element. We have $\gamma \geq 3\tau - 1$, so $T[a\,..\,a+2\tau)$ is a synchronizing fragment that occurs in P at position $b = a - j$. By the consistency condition of synchronizers, no synchronizing fragment of T occurs in P_0 at a position smaller than $b - i_0\gamma$. Thus, $b \in B_1$. Then $j = a - b \in A_1 \ominus B_1$, as required.

Now assume that $p := \mathsf{per}(P_0) \leq \tau/3$. Then $T[j + i_0\gamma\,..\,j + (i_0+1)\gamma)$ extends uniquely to a run $T[x\,..\,y]$ with period p. As $\gamma \geq 3\tau - 1$, this run is a τ-run. Let $A'_2 = \mathsf{Misper}_{k+1}(T, x, x + p) = \mathsf{Misper}_{k+1}(T, j + i_0\gamma, j + i_0\gamma + p)$ and $B'_2 = \mathsf{Misper}_{k+1}(P, i_0\gamma, i_0\gamma + p)$. Assume first that $(A'_2 \ominus j) \cap B'_2 \neq \emptyset$ and let q be an element of this set. By definition, $A'_2 \subseteq A_2$ and $B'_2 \subseteq B_2$. Then $j = (q + j) - q \in A_2 \ominus B_2$, as desired.

Finally, assume that $(A'_2 \ominus j) \cap B'_2 = \emptyset$. We note that $A'_2 \ominus j \supseteq \mathsf{Misper}_{k+1}(T', i_0\gamma, i_0\gamma + p)$ for $T' = T[j\,..\,j+m)$ and recall that $B'_2 = \mathsf{Misper}_{k+1}(P, i_0\gamma, i_0\gamma + p)$. By Lemma 6.3, $\mathsf{Misper}_{k+1}(T', i_0\gamma, i_0\gamma + p) = \mathsf{Misper}(T', i_0\gamma, i_0\gamma + p)$, $B'_2 = \mathsf{Misper}(P, i_0\gamma, i_0\gamma + p)$, and $\delta_H(P, T') = |\mathsf{Misper}(T', i_0\gamma, i_0\gamma + p)| + |\mathsf{Misper}(P, i_0\gamma, i_0\gamma + p)|$. Hence, T' is a k-nearly periodic occurrence of P. □

We deal with nearly periodic occurrences using properties of runs in the text. A proof of the following lemma is deferred until the end of the section.

Lemma 6.8. *For a text T of length n over an integer alphabet and $k = \mathcal{O}(1)$, there is an index of size $\mathcal{O}(n)$ that, given a pattern P of length m, reports a set of k-mismatch occurrences of P in*

T that contains all k-nearly periodic occurrences in $\mathcal{O}(\log\log n + m + \mathsf{occ})$ time where occ is the number of k-mismatch occurrences of P.

In the 2D orthogonal range reporting queries problem, we are to construct a data structure over a specified set of points in a 2D grid that supports the following queries: given a rectangle, list all points that belong to the rectangle. We use the following data structure.

Theorem 6.9 (Alstrup et al. [ABR00, Theorem 2])**.** *For a set of n points in an $n \times n$ grid, there exists a data structure of size $\mathcal{O}(n \log^{\varepsilon} n)$, for any constant $\epsilon > 0$, that answers orthogonal range reporting queries in $\mathcal{O}(t + \log\log n)$ time, where t is the number of reported points.*

Let us restate the main result of this section for convenience.

Theorem 1.5. *For a text T of length n, positive integers $k = \mathcal{O}(1)$, $\gamma \in [2\,..\,n]$, and constant $\varepsilon > 0$, there exists an $\mathcal{O}(n + n\log^{k+\varepsilon} n/\gamma)$-space k-mismatch index that answers k-mismatch queries for patterns of length $m \geq (k+1)\gamma$ in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time.*

Proof. Nearly periodic k-mismatch occurrences are handled by the data structure of Lemma 6.8. The query complexity of that data structure is dominated by the final query complexity. The remaining k-mismatch occurrences are handled using the anchors from Lemma 6.7.

Data structure. The index uses k-errata trees $\mathcal{T}_t$ for $t \in \{1,2\}$ for the sets of suffixes $\{T[i\,..\,n) : i \in A_t\}$ of T and $\mathcal{T}'_t$ for $t \in \{1,2\}$ for the sets of suffixes $\{(T[0\,..\,i))^R : i \in A_t\}$ of T^R.

Let us recall that for $k_1 \in [0\,..\,k)$, a k_1-errata tree is formed by compact tries of the k-errata tree at levels in $[0\,..\,k_1]$. Let $\mathcal{T}_{t,k_1}$ be the k_1-errata tree formed from $\mathcal{T}_t$. For every $t \in \{1,2\}$ and $k_1 \in [0\,..\,k]$, let S_{t,k_1} be a concatenation of labels of terminals in all compact tries in $\mathcal{T}_{t,k_1}$. In each compact trie, the terminals are concatenated from left to right. The order of compact tries can be arbitrary. Each explicit node of each compact trie in $\mathcal{T}_{t,k}$ stores the fragment of S_{t,k_1} that corresponds to the terminal labels in its subtree. Similarly, we let $\mathcal{T}'_{t,k_2}$ for $k_2 \in [0\,..\,k]$ be the k_2-errata tree formed from $\mathcal{T}'_t$ and S'_{t,k_2} be a concatenation of labels of terminals in all compact tries in $\mathcal{T}'_{t,k_2}$.

For every $t \in \{1,2\}$ and $k_1 \in [0\,..\,k]$, we construct a 2D orthogonal range reporting data structure for the set of 2D points $\{(x,y) : x \in [0\,..\,|S_{t,k_1}|), y \in [0\,..\,|S'_{t,k_2}|), S_{t,k_1}[x] = n - S'_{t,k_2}[y]\}$ where $k_2 = k - k_1$.

Space complexity. By Lemma 6.4, the sizes of the sets of anchors A_1, A_2 are $\mathcal{O}(n/\gamma)$. Hence, by Theorem 5.1, the sizes of the k-errata trees are $\mathcal{O}(n + n\log^k n/\gamma)$.

By Lemma 5.2, for each $k_1 \in [0\,..\,k]$, string S_{t,k_1} contains $\mathcal{O}(\log^{k_1} n)$ copies of each of the $\mathcal{O}(n/\gamma)$ characters. Similarly, string S'_{t,k_2} for $k_2 = k - k_1$ contains $\mathcal{O}(\log^{k_2} n)$ copies of every character. Over all k_1, the number of points created in the 2D orthogonal range reporting data structure is $\mathcal{O}(n\log^k n/\gamma)$. By Theorem 6.9, this gives $\mathcal{O}(n\log^{k+\varepsilon} n/\gamma)$ space of the range reporting data structures.

Queries. Given a pattern P of length $m \geq (k+1)\gamma$, we construct its sets of anchors B_1 and B_2 using Lemma 6.5. For each $t \in \{1,2\}$, each $k_1 \in [0\,..\,k]$ and each anchor $b \in B_t$, we ask a query for the suffix $P[b\,..\,m)$ to the k_1-errata tree $\mathcal{T}_{t,k_1}$ and a query for the suffix $(P[0\,..\,b))^R$ of P^R to the k_2-errata tree $\mathcal{T}'_{t,k_2}$ where $k_2 = k - k_1$. In each query, we do *not* report all occurrences. Instead, each query results in a set of nodes in $\mathcal{T}_{t,k_1}$ which translate to fragments of S_{t,k_1} and a set of nodes in $\mathcal{T}'_{t,k_2}$ which translate to fragments of S'_{t,k_2}. For every fragment $S_{t,k_1}[x_1\,..\,x_2]$ and every fragment $S'_{t,k_2}[y_1\,..\,y_2]$, we ask an orthogonal range reporting query for the rectangle $[x_1\,..\,x_2] \times [y_1\,..\,y_2]$ in the data structure constructed for t and k_1. For every reported point (x,y), we report a k-mismatch occurrence of P in T at position $S_{t,k_1}[x] - b$.

Query complexity. First, we analyze all steps excluding reporting of occurrences. The sets of anchors B_1, B_2 are constructed in $\mathcal{O}(m)$ time. For a given anchor, we ask $\mathcal{O}(k)$ queries to $(\leq k)$-errata trees. After $\mathcal{O}(m)$ time preprocessing, a query for k_1 returns in $\mathcal{O}(\log^{k_1} n \log\log n)$ time $\mathcal{O}(\log^{k_1} n)$ nodes in $\mathcal{T}_{t,k_1}$ and a query for k_2 returns in $\mathcal{O}(\log^{k_2} n \log\log n)$ time $\mathcal{O}(\log^{k_2} n)$ nodes in $\mathcal{T}'_{t,k_2}$. Overall, in $\mathcal{O}(\log^{k} n \log\log n)$ time we obtain $\mathcal{O}(\log^{k} n)$ queries to a 2D orthogonal range reporting data structure, for a total of $\mathcal{O}(\log^{k} n \log\log n)$ time per anchor. The time complexity follows by Theorem 6.9.

Finally, let us count how many times a given k-mismatch occurrence of P in T can be reported. For every $k_1 \in [0\,..\,k]$, $k_2 = k - k_1$, $t \in \{1, 2\}$, and $b \in B_t$, each k_1-mismatch occurrence of $P[b\,..\,m)$ in T at a position in A_t is considered in exactly one interval and each k_2-mismatch occurrence of $P[0\,..\,b)$ in T ending at a position in $A_t \ominus 1$ is also considered exactly once. Hence, for fixed k_1, t, and b, each resulting k-mismatch occurrence of P will be reported at most once. Over all k_1 and b, each occurrence is reported $\mathcal{O}(k^3) = \mathcal{O}(1)$ times, by Lemma 6.5. □

We say that a string U is *primitive* if $U = V^t$ for string V and positive integer t implies that $t = 1$. Strings U and V are called *cyclic shifts* if there exist strings X, Y such that $U = XY$ and $V = YX$. A *Lyndon string* is a string U that is primitive and is lexicographically minimal among its cyclic shifts. All cyclic shifts of a primitive string are primitive and different [Lot97]. Hence, every primitive string U has a unique cyclic shift V (the minimal one) that is a Lyndon string. We extend this notation to runs as follows: a Lyndon root of a periodic string U with period p is the minimum cyclic shift of the string period $U[0\,..\,p)$ of U (see [CIK+14]).

The problem of interval stabbing is defined as follows: Preprocess a set of n intervals so that, given a query point a, all intervals that contain the point a can be reported efficiently. Interval stabbing queries enjoy a known reduction to three-sided (actually, even two-sided) 2D orthogonal range reporting queries. An interval $[\ell\,..\,r]$ is treated as a point (ℓ, r) in 2D and a stabbing query for point a requires to compute all points in a rectangle $(-\infty\,..\,a] \times [a\,..\,\infty)$. Using the data structure for three-sided 2D orthogonal range reporting of Alstrup et al. [ABR00], we obtain the following data structure for interval stabbing.

Theorem 6.10 (see [ABR00, Corollary 1])**.** *A set of n intervals with endpoints in $[0\,..\,N]$ can be preprocessed into a data structure of size $\mathcal{O}(n)$ that answers interval stabbing queries in $\mathcal{O}(t + \log\log N)$ time, where t is the number of reported intervals.*

We are ready to prove Lemma 6.8.

Proof of Lemma 6.8. Data structure. The data structure contains a trie $\mathcal{M}$ of all strings being a Lyndon root of a τ-run in T. Each terminal node of $\mathcal{M}$, representing a substring of T that is a Lyndon root U of some τ-runs in T, holds $(k+1)|U|$ interval stabbing data structures indexed by pairs in $[0\,..\,k] \times [0\,..\,|U|)$ that we now describe.

Let $T[x\,..\,y]$ be a τ-run in T with period p and $Q = T[d\,..\,d+p)$ for $d \in [x\,..\,x+p)$ be its Lyndon root. We consider the sets $L = \mathsf{LeftMisper}_{k+1}(T, x, x+p)$ and $R = \mathsf{RightMisper}_{k+1}(T, x, x+p)$. If $|L| \leq k$, we insert a sentinel position -1 to L. Similarly, if $|R| \leq k$, we insert a sentinel position n to R. Let $L = \{\ell_1, \ell_2, \ldots, \ell_{k_1}\}$ with $\ell_{k_1} < \ell_{k_1-1} < \cdots < \ell_1$ $(k_1 \leq k+1)$. Let $R = \{r_1, r_2, \ldots, r_{k_2}\}$ with $r_1 < r_2 < \cdots < r_{k_2}$ $(k_2 \leq k+1)$. We set $r_0 = \ell_1$.

Let us consider each $a \in [1\,..\,k_1)$ and $b \in [0\,..\,k_2)$ such that $a + b \leq k$ and each remainder $t \in [0\,..\,p)$. Let α and β be the minimum and maximum element of the set $\{j \in (\ell_{a+1}\,..\,\ell_a] : d - j \equiv t \pmod p\}$, if the set is non-empty. We then insert an interval $I = [r_b - \beta + 1\,..\,r_{b+1} - \alpha]$ to the data structure corresponding to the Lyndon root Q indexed by the pair $(a+b, t)$. Intuitively, the interval I has the following interpretation: for every m in I, there exists an index $j \in [\alpha\,..\,\beta]$ such that $j \equiv \alpha$

(mod p) and $j+m-1 \in [r_b\,..\,r_{b+1})$. Then for $T' = T[j\,..\,j+m)$, $|\mathsf{Misper}(T', d-j, d-j+p)| = a+b$, where $T'[d-j\,..\,d-j+p) = Q$, and $d-j \equiv t \pmod p$. The interval I in the data structure stores additional data: numbers α, β as well as the interval $[\alpha'\,..\,\beta'] = [r_b\,..\,r_{b+1})$. Formally, the additional data can be stored using perfect hashing [FKS84]. With the additional data, we will be able to recover all values $j \in [\alpha\,..\,\beta]$ such that $j \equiv \alpha \pmod p$ and $j+m-1 \in [\alpha'\,..\,\beta']$ using simple arithmetics.

The case in which a pattern P can match a length-m periodic substring of T is handled similarly. For each remainder $t \in [0\,..\,p)$, we compute the minimum α and the maximum β of the set $\{j \in (\ell_1\,..\,r_1) : d-j \equiv t \pmod p\}$ and insert an interval $[0\,..\,r_1-\alpha]$ to the data structure corresponding to the Lyndon root Q of the run $T[x\,..\,y]$ indexed by the pair $(0, d)$. The additional data stored with the interval are α, β, and $\beta' = r_1$.

We also store the data structure of Lemma 2.4 for T.

Data structure size. By Lemma 6.2, the total number of τ-runs in T is $\mathcal{O}(n/\tau)$. The length of each Lyndon root of a τ-run is at most $\tau/3$, so the total size of the trie $\mathcal{M}$ is $\mathcal{O}(n)$, even without compactification. There are $\mathcal{O}((n/\tau)\cdot k\cdot\tau) = \mathcal{O}(n)$ interval stabbing data structures. For each τ-run, we consider $\mathcal{O}(k^2)$ pairs of indices (a, b) and $\mathcal{O}(\tau)$ remainders t and for each a, b, t insert one interval into a data structure. The interval stabbing data structures contain $\mathcal{O}(k^2 n)$ intervals in total and thus, by Theorem 6.10, can be represented in $\mathcal{O}(n)$ total space. The data structure of Lemma 2.4 takes just $\mathcal{O}(n)$ space.

Queries. Let P be a query pattern of length $m \geq (k+1)\gamma$. For every $i \in [0\,..\,k]$, we compute the smallest period p of $P[i\gamma\,..\,(i+1)\gamma)$. If $p \leq \tau/3$ and $P[i\gamma\,..\,(i+1)\gamma)$ is a substring of T, which we check using Lemma 2.4, we proceed as follows. We compute the set $X = \mathsf{Misper}_{k+1}(P, i\gamma, i\gamma+p)$ by definition. If $|X| \leq k$, we proceed as follows. The Lyndon root $Q = P[d\,..\,d+p)$, with $d \in [i\gamma\,..\,i\gamma+p)$, of $P[i\gamma\,..\,i\gamma+p)$ is computed using an algorithm for a minimal cyclic shift [Boo80, Duv83]. We then check if Q is a terminal in $\mathcal{M}$ using Theorem 2.2. If so, we query the interval stabbing data structures for Q indexed by pairs $(h, d \bmod p)$ for all $h \in [0\,..\,k-|X|]$ for the length m. Assume that $h > 0$ and the data structure returned an interval I ($m \in I$) that stores, as additional data, numbers α, β as well as the interval $[\alpha'\,..\,\beta']$. We then report a k-mismatch occurrence of P in T at each index $j \in [\alpha\,..\,\beta]$ such that $j \equiv \alpha \pmod p$ and $m \in [\alpha'+1-j\,..\,\beta'+1-j]$; the final condition can be restated equivalently as $j \in [\alpha'+1-m\,..\,\beta'+1-m]$. There will be at least one such index j by the fact that the interval I was stabbed by m. For $h = 0$, if the data structure returned an interval with additional data α, β, and β', we report all $j \in [\alpha\,..\,\beta]$ such that $j \equiv \alpha \pmod p$ and $j+m-1 \leq \beta'$. They will be initial elements of the sequence $\alpha, \alpha+p, \ldots, \beta$.

Query correctness. Let us argue for the correctness of the query algorithm. First, we show that if position j is reported in the algorithm, then $\delta_H(T[j\,..\,j+m), P) \leq k$, i.e. j is the starting position of a k-mismatch occurrence of P in T. Assume that j is reported for substring $P[i\gamma\,..\,(i+1)\gamma)$ for $i \in [0\,..\,k]$, with smallest period p and Lyndon root $Q = T[d\,..\,d+p)$, when querying the data structure indexed by pair $(h, d \bmod p)$ for $h \in [0\,..\,k-|X|]$, where $X = \mathsf{Misper}_{k+1}(P, i\gamma, i\gamma+p)$, $|X| \leq k$. Let $W = Q^\infty[c\,..\,c+m)$ for $c = (-d) \bmod p$. We have $\delta_H(W, P) = |X|$ and $\delta_H(W, T[j\,..\,j+m)) = h \leq k-|X|$. By the triangle inequality, $\delta_H(P, T[j\,..\,j+m)) \leq \delta_H(W, P) + \delta_H(W, T[j\,..\,j+m)) \leq k$, as required.

Now we show that if $T' = T[j\,..\,j+m)$ is a k-nearly periodic occurrence of P in T, then j is reported in the algorithm. By definition, there is $i \in [0\,..\,k]$ such that $P[i\gamma\,..\,(i+1)\gamma)$ has smallest period $p \leq \tau/3$. Let $d \in [i\gamma\,..\,i\gamma+p)$ be such that $Q = P[d\,..\,d+p)$ is the Lyndon root of $P[i\gamma\,..\,i\gamma+p)$. By definition, $P[i\gamma\,..\,(i+1)\gamma) = T'[i\gamma\,..\,(i+1)\gamma)$. The fragment $T[j+i\gamma\,..\,j+(i+1)\gamma)$ extends to a run $T[x\,..\,y]$ in T with period p; it is a τ-run as $\gamma \geq 3\tau-1$ and $p \leq \tau/3$. Let $X = \mathsf{Misper}(P, i\gamma, i\gamma+p)$ and $Y = \mathsf{Misper}(T', i\gamma, i\gamma+p)$. By definition, $|X|+|Y| \leq k$. We have $(Y \oplus j) \subseteq (L \cup R)$ where $L = \mathsf{LeftMisper}_{k+1}(T, x, x+p)$ and $R = \mathsf{RightMisper}_{k+1}(T, x, x+p)$. Then

j is reported for $a = |L \cap (Y \oplus j)|$, $b = |R \cap (Y \oplus j)|$, and the remainder $t = d \bmod p$.

Query complexity. The smallest period of the fragment $P[i\gamma\,..\,(i+1)\gamma)$ is computed in $\mathcal{O}(\gamma)$ time [KJP77], for a total of $\mathcal{O}(m)$ time over $i \in [0\,..\,k]$. We check if $P[i\gamma\,..\,(i+1)\gamma)$ is a substring of T in $\mathcal{O}(1)$ time after $\mathcal{O}(m)$ preprocessing of Lemma 2.4. The Lyndon roots for all the fragments are also computed in $\mathcal{O}(m)$ total time using a linear-time algorithm [Boo80, Duv83]. All the sets $\mathsf{Misper}_{k+1}(P, i\gamma, i\gamma + p)$ for $i \in [0\,..\,k]$ such that $P[i\gamma\,..\,(i+1)\gamma)$ has period $p \le \tau/3$ are computed in $\mathcal{O}(mk)$ time by definition. The trie $\mathcal{M}$ is descended for U in $\mathcal{O}(\tau)$ time (Theorem 2.2), for a total of $\mathcal{O}(m)$ time. Then, for a given i we ask $\mathcal{O}(k)$ interval stabbing queries, which requires $\mathcal{O}(k^2 \log\log n)$ time in addition to the time for reporting occurrences.

Finally, let us show that each k-mismatch occurrence $T[j\,..\,j+m)$ of P is reported at most $\mathcal{O}(k^2)$ times by the data structure. First, it can be reported for each fragment $P[i\gamma\,..\,(i+1)\gamma)$ with $i \in [0\,..\,k]$. This fragment implies exactly one Lyndon root Q. If $T[j\,..\,j+m)$ has period p, it is reported for exactly one τ-run in T with Lyndon root Q, namely, for the τ-run that extends $T[j\,..\,j+m)$. In total, if $T[j\,..\,j+m)$ has period p, it is reported $\mathcal{O}(k)$ times for $i \in [0\,..\,k]$.

Henceforth, we assume that $T[j\,..\,j+m)$ does not have period p. For a given τ-run $T[x\,..\,y]$ in T with Lyndon root Q, at most one interval I is inserted to an interval stabbing data structure such that $m \in I$, $j \in [\alpha\,..\,\beta]$, $j \equiv \alpha \pmod p$, and $j + m - 1 \in [\alpha'\,..\,\beta']$ where α, β, and $[\alpha'\,..\,\beta']$ are the additional data stored for the interval I. It suffices to note that $T[j\,..\,j+m)$ will be reported by at most k τ-runs with Lyndon root Q in T. Indeed, each such τ-run $T[x\,..\,y]$ must satisfy $x > j$. Let $T[x_1\,..\,y_1], \ldots, T[x_{k'}\,..\,y_{k'}]$ be all such τ-runs ordered by increasing starting positions. (Two runs with period p can intersect on at most $p-1$ positions; otherwise they would be the same run.) By maximality of runs, for any $k'' > k$, $\mathsf{LeftMisper}(T, x_{k''}, x_{k''} + p)$ contains at least $k+1$ elements that are at least j. This means that $T[j\,..\,j+m)$ will not be reported by $T[x_{k''}\,..\,y_{k''}]$.

This concludes that the query complexity is $\mathcal{O}(m + \log\log n + \mathsf{occ})$. □

7 $\mathcal{O}(n \log^{k-1} n)$-space k-Approximate Index

We improve upon the approach of Chan et al. [CLS+10] to make it work for a text over any alphabet within the same complexity. First we show a simpler data structure for (exact) indexing with up to k wildcards in the pattern, as it highlights our approach better. Then we show how it can be adjusted to k-Mismatch Indexing.

7.1 Index for Wildcards in the Pattern

A wildcard is a don't care symbol that is not in the alphabet of the text or the pattern. A wildcard matches every character of the alphabet; the relation of matching extends naturally to strings with wildcards. We improve the space complexity of an index for pattern matching queries for patterns with up to k wildcards in a string text as shown in Table 3.

7.1.1 Original k-errata tree for wildcards in the pattern

Cole et al. [CGL04] proposed the following version of k-errata tree for indexing patterns with wildcards.

Theorem 7.1 ([CGL04])**.** *Let T be a string text of length n. The k-errata tree for T uses space $\mathcal{O}(n \log^k n)$ and, for a pattern P of length m with up to k wildcards, answers a pattern matching query for P in $\mathcal{O}(2^k \log\log n + m + \mathsf{occ})$ time, where occ is the number of occurrences of P in T.*

Space	Query time (plus $\mathcal{O}(m+\mathsf{occ})$)	Reference	Comment
$\mathcal{O}(n\sigma^{k^2}(\log\log n)^k)$	$\mathcal{O}(k)$	[BGVV14]	
$\mathcal{O}(n^{k+1})$	$\mathcal{O}(k)$	[BGVV14]	
$\mathcal{O}(n\log^k n)$	$\mathcal{O}(2^k\log\log n)$	[CGL04]	improved here
$\mathcal{O}(n\log^{k-1} n)$		**Thm. 1.3**	
$\mathcal{O}(n\log n(\log_\alpha n)^{k-1})$	$\mathcal{O}(\alpha^k\log\log n)$	[BGVV14]	
$\mathcal{O}(n\log n)$	$\mathcal{O}(\sigma^k\log\log n)$	[CGL04]	improved by [BGVV14]
$\mathcal{O}(n)$	$\mathcal{O}(\sigma^k\min\{m,\log\log n\})$	[BGVV14]	

Table 3: Indexing for patterns with up to k wildcards

The recursive structure of the k-errata tree in this case is very similar as for mismatches; only the details on how compact tries at subsequent levels are formed are different (actually, simpler). More precisely, all substitution trees of the type (a), i.e., off-path subtrees $\mathcal{T}^s_{v,a}$, are organized into a single group tree, whereas substitution trees and group trees of type (b) are not formed. Thus, each group tree can be viewed as a sparse suffix tree of T (technically, with a wildcard-character edge as in item (1) in Section 5).

The query algorithm also has the same recursive structure, but it is simpler, as here the positions of wildcards are known, whereas the positions of mismatches were not known. Here, if the query algorithm is called recursively on a compact trie $\mathcal{T}$ with a suffix P' of the pattern P, then the level of the trie plus the number of wildcards in P' does not exceed k. The query time is $\mathcal{O}(2^k\log\log n)$.

We show how to improve the space complexity from $\mathcal{O}(n\log^k n)$ to $\mathcal{O}(n\log^{k-1} n)$ retaining the query complexity. We note that this variant was not considered by Chan et al. [CLS+10] as their data structure does not give a significant improvement here. Particularly, it would give space $\mathcal{O}(n\log^{k-1} n)$ but make the query time $\mathcal{O}(\sigma\cdot 2^k\log\log n)$.

7.1.2 Storing suffixes at level k compactly

We save space by explicitly constructing only the $(k-1)$-errata tree. The final step of the recursion is replaced by storing arrays equivalent to group trees at level k compactly.

Let $\mathcal{T}$ be a compact trie at level $k-1$ of the errata tree. For a node u of $\mathcal{T}$, let $\mathcal{T}_u$ be the subtree of $\mathcal{T}$ rooted at u. By $|\mathcal{T}_u|$ we denote the number of terminals in $\mathcal{T}_u$ (these are the terminals from $\mathcal{T}$ that are located in $\mathcal{T}_u$). We use the heavy-light decomposition of $\mathcal{T}$. For an explicit node v of $\mathcal{T}$, let $u_1,\ldots,u_t$ be all explicit children of v that are not located on the same heavy path as v. Let us create a list of path labels of all terminals of the tree $\mathcal{T}_v$ that are located in the subtrees $\mathcal{T}_{u_1},\ldots,\mathcal{T}_{u_t}$, trim their first characters (that immediately follow v), and order the resulting strings lexicographically. We denote the list of resulting suffixes of T as $S'_{v,1},\ldots,S'_{v,b_v}$. Furthermore, by $S_{v,1},\ldots,S_{v,b_v}$ we denote the list of these suffixes without their first character removed, but *in the order* of their truncated versions $S'_{v,1},\ldots,S'_{v,b_v}$.

Over all compact tries at level $k-1$ of the errata tree, the lists of trimmed suffixes have total length $\mathcal{O}(n\log^k n)$ (as each terminal in $\mathcal{T}$ generates suffixes in $\mathcal{O}(\log n)$ nodes v, by the properties of heavy paths). Instead of storing them explicitly, we will store the indices

$$label'_v[i] := n-1-|S'_{v,i}|$$

of the trimmed suffixes using just $\mathcal{O}(n\log^k n)$ bits of space in total, i.e., $\mathcal{O}(n\log^{k-1} n)$ words of space. This will be done using similar techniques as in Chan et al. [CLS+10].

Let v be an explicit node of a compact trie $\mathcal{T}$ at level $k-1$ and $u_1, \ldots, u_t$ be all the explicit children of v not on the same heavy path. We assign subsequent *ranks* $1, 2, \ldots$ to the subtrees $\mathcal{T}_{u_i}$ so that trees with more terminals receive a smaller rank, breaking ties arbitrarily. We store the following arrays:

- $\mathit{st\text{-}rank}'_v[1\,..\,b_v]$: If $S_{v,i}$ is a terminal of $\mathcal{T}_v$ located in $\mathcal{T}_{u_j}$, $\mathit{st\text{-}rank}'_v[i] \in [1\,..\,t]$ is the rank of $\mathcal{T}_{u_j}$.
- $\mathit{tree\text{-}pointer}'_v[1\,..\,t]$: $\mathit{tree\text{-}pointer}'_v[j]$ points to the subtree $\mathcal{T}_{u_i}$ with rank j.
- $\mathit{modified\text{-}rank}'_{v,u_a}[1\,..\,|\mathcal{T}_{u_a}|]$: $\mathit{modified\text{-}rank}'_{v,u_a}[j] = i \in [1\,..\,b_v]$ if the jth lexicographic terminal in $\mathcal{T}_{u_a}$ generates $S'_{v,i}$.

Remark 7.2. In [CLS+10], similar arrays *st-rank*, *tree-pointer*, *modified-rank* were used but in a different context; see Section 7.2.

The proof of next lemma resembles the proof of [CLS+10, Lemma 2].

Lemma 7.3. *The sum of values* $\lceil \log \mathit{st\text{-}rank}'_v[i] \rceil$, *over all explicit nodes* v *in compact tries* $\mathcal{T}$ *at level* $k-1$ *and* $i \in [1\,..\,b_v]$, *is* $\mathcal{O}(n \log^k n)$.

Proof. Let ℓ be a terminal in a compact trie $\mathcal{T}$ at level $k-1$. We will sum up values $\lceil \mathit{st\text{-}rank}'_v[i] \rceil$ for v, i such that $S_{v,i}$ corresponds to ℓ.

Let $C_1, C_2, \ldots, C_\alpha$ be all heavy paths, top-down, visited on the root-to-ℓ path π. By $\mathcal{T}_{C_i}$ we denote the subtree of $\mathcal{T}$ rooted at the root of path C_i. For $j \in [1\,..\,\alpha)$, let v be a node in heavy path C_j where the path π leaves the path C_j, u_a be the next explicit node on π, and let r_j be the rank of $\mathcal{T}_{u_a}$. By the definition of rank,

$$r_j \;\leq\; \frac{|\mathcal{T}_v|}{|\mathcal{T}_{u_a}|} \;\leq\; \frac{|\mathcal{T}_{C_j}|}{|\mathcal{T}_{u_a}|} \;=\; \frac{|\mathcal{T}_{C_j}|}{|\mathcal{T}_{C_{j+1}}|}.$$

We thus have

$$\sum_{j=1}^{\alpha-1} \log r_j \;\leq\; \sum_{j=1}^{\alpha-1} \log \frac{|\mathcal{T}_{C_j}|}{|\mathcal{T}_{C_{j+1}}|} \;\leq\; \log |\mathcal{T}_{C_1}| \;\leq\; \log n.$$

We notice that for every node v and index $i \in [1\,..\,b_v]$, $\mathit{st\text{-}rank}'_v[i]$ uniquely corresponds to r_j for ℓ being the locus of $L_v S_{v,i}$ where L_v is the path label of v and j such that the root-to-ℓ path leaves the jth heavy path C_j at node v. By Theorem 7.1, there are $\mathcal{O}(n \log^{k-1} n)$ terminals in compact tries at level $k-1$, so the sum of all values $\log \mathit{st\text{-}rank}'_v[i]$ is $\mathcal{O}(n \log^k n)$. The sum of values $\lceil \log \mathit{st\text{-}rank}'_v[i] \rceil$ is also $\mathcal{O}(n \log^k n)$ as the number of these values is $\mathcal{O}(n \log^k n)$. □

By Lemma 7.3, the *st-rank*′ arrays can be represented in $\mathcal{O}(n \log^k n)$ bits using variable size encoding that allows $\mathcal{O}(1)$-time access [Mun96]. The *tree-pointer*′ arrays contain $\mathcal{O}(n \log^{k-1} n)$ values in total, one per each compact edge of a level-$(k-1)$ compact trie. Finally, *modified-rank*′ arrays can be stored efficiently according to the next lemma that resembles [CLS+10, Lemma 11].

Lemma 7.4. *All* $\mathit{modified\text{-}rank}'$ *arrays can be stored in* $\mathcal{O}(n \log^k n)$ *bits of space so that a (restricted) rank query on an array* $\mathit{modified\text{-}rank}'_{v,u_a}$ *can be answered in* $\mathcal{O}(1)$ *time.*

Proof. Let v be an explicit node in a compact trie of level $k-1$ and $u_1, \ldots, u_t$ be its explicit children located on a different heavy path. Recall that in a rank query, given array $\mathit{modified\text{-}rank}'_{v,u_a}$ and a value i, we are to retrieve the index j such that $\mathit{modified\text{-}rank}'_{u_a}[j] = i$ or state that j does not exist. The sequence $\mathit{modified\text{-}rank}'_{v,u_a}$ is increasing, has length $|\mathcal{T}_{u_a}|$ and contains elements in $[1\,..\,|\mathcal{T}_v|]$,

so by Theorem 2.1, it can be represented in $\mathcal{O}\left(|T_v|\left\lceil\log\frac{|\mathcal{T}_v|}{|\mathcal{T}_{u_a}|}\right\rceil\right)$ bits of space to allow rank queries. We have $\mathcal{O}\left(|T_v|\left\lceil\log\frac{|\mathcal{T}_v|}{|\mathcal{T}_{u_a}|}\right\rceil\right)=\mathcal{O}\left(|T_v|\log\frac{|\mathcal{T}_v|}{|\mathcal{T}_{u_a}|}\right)$ as $|\mathcal{T}_{u_a}|\leq\frac{1}{2}|\mathcal{T}_v|$ (the child u_a is light).

Let $f(\mathcal{T}_v)$ be the total space in bits required to store the *modified-rank′* arrays for all valid pairs of nodes in the subtree $\mathcal{T}_v$. We have

$$f(\mathcal{T}_v)\leq\sum_{i=1}^{t}\mathcal{O}\left(|\mathcal{T}_{u_i}|\log\frac{|\mathcal{T}_v|}{|\mathcal{T}_{u_i}|}\right)+f(\mathcal{T}_{u_i})$$

which solves to $f(\mathcal{T}_v)=\mathcal{O}(|\mathcal{T}_v|\log|\mathcal{T}_v|)$ for any node v. By taking as nodes v the roots of all compact tries at level $k-1$, we obtain the desired bound. □

Finally, an array called a sparse suffix array storing a lexicographic list of all terminals of $\mathcal{T}$ is stored, with each terminal represented as the position in T of the corresponding suffix. Each explicit node u of $\mathcal{T}$ stores a fragment of the list that corresponds to the terminals of $\mathcal{T}$ that are located in $\mathcal{T}_u$. The total size of the sparse suffix arrays stored is $\mathcal{O}(n\log^{k-1}n)$.

In the next lemma, we show that the arrays are sufficient to store labels of the trimmed suffixes.

Lemma 7.5. *Let v be an explicit node in a compact trie $\mathcal{T}$ at level $k-1$ of the errata tree. Assuming random access to the sparse suffix array of $\mathcal{T}$, arrays st-rank$'_v$ and tree-pointer$'_v$, and $\mathcal{O}(1)$-time rank queries on all arrays modified-rank$'_{v,u_a}$, given i, label$'_v[i]$ can be computed in $\mathcal{O}(1)$ time.*

Proof. The original suffix $S_{v,i}$ that generates the trimmed suffix $S'_{v,i}$ originates from the trie $\mathcal{T}_{u_a}=$ *tree-pointer*$'_v$[*st-rank*$'_v[i]$] that can be retrieved in $\mathcal{O}(1)$ time. We compute an index j such that *modified-rank*$'_{v,u_a}[j]=i$ using a rank query in $\mathcal{O}(1)$ time. The suffix represented by the jth terminal in $\mathcal{T}_{u_a}$ can be extracted from the jth element in the fragment of the sparse suffix array for $\mathcal{T}$ that corresponds to $\mathcal{T}_{u_a}$. From the suffix we retrieve $S'_{v,i}$ which implies *label*$'_v[i]$. □

7.1.3 Compact k-errata tree for wildcards in the pattern

The key improvements that allow to achieve a more compact data structure are the developments of Section 7.1.2 as well as sampling suffixes among $S'_{v,1},\ldots,S'_{v,b_v}$ according to Lemma 4.5.

Theorem 1.3. *For a string T of length n, there exists an index using $\mathcal{O}(n\log^{k-1}n)$ space that answers pattern matching queries for patterns of length m with up to k wildcards in $\mathcal{O}(2^k\log\log n+m+\mathsf{occ})$ time.*

Proof. Data structure. As mentioned before, we store the $(k-1)$-errata tree explicitly together with the data structure for answering unrooted TreeLCP queries in compact tries (Theorem 2.3). By Theorem 7.1, this takes $\mathcal{O}(n\log^{k-1}n)$ space. For all explicit nodes v of compact tries at level $k-1$, we store (1) the array *st-rank*$'_v$ using variable size encoding [Mun96], which take $\mathcal{O}(n\log^{k-1}n)$ total space by Lemma 7.3; (2) the *tree-pointer*$'_v$ array in $\mathcal{O}(n\log^{k-1}n)$ total space; and (3) the *modified-rank*$'_{v,u_a}$ arrays for all u_a with rank queries using Lemma 7.4 also in $\mathcal{O}(n\log^{k-1}n)$ total space. We also store the sparse suffix array of each compact trie $\mathcal{T}$ at level $k-1$.

For each node v of each compact trie $\mathcal{T}$ at level $k-1$ of the data structure, let $\mathcal{T}'_v$ denote the compact trie of trimmed suffixes $S'_{v,1},\ldots,S'_{v,b_v}$. We select every $\lfloor\log n\rfloor$th trimmed suffix among $S'_{v,1},\ldots,S'_{v,b_v}$, as well as $S'_{v,1}$ and S'_{v,b_b}, and construct a compact trie $\mathcal{T}''_v$ of these trimmed suffixes. Each terminal $S'_{v,i}$ in $\mathcal{T}''_v$ stores i as its number. The total size of the compact tries $\mathcal{T}''_v$ is $\mathcal{O}(n\log^{k-1}n)$, whereas trees $\mathcal{T}'_v$ are *not* stored explicitly. We construct a data structure for answering unrooted TreeLCP queries on each of $\mathcal{T}''_v$ (Theorem 2.3), which does not affect the space complexity.

We also store the data structure of Lemma 2.4 together with the data structure for LCP-queries in T [BF00]; both take just $\mathcal{O}(n)$ space.

Queries. We execute a $(k-1)$-errata tree query for P (Theorem 7.1). As a result, in time $\mathcal{O}(2^k \log\log n)$, $\mathcal{O}(2^{k-1})$ pairs $(\mathcal{T}, P')$ are computed where $\mathcal{T}$ is a compact trie at level $k-1$ and P' is a suffix of P with at most 1 wildcard; our goal is to match P' against terminals in $\mathcal{T}$. We compute $v = \mathsf{TreeLCP}(\mathcal{T}, P')$ (in $\mathcal{O}(\log\log n)$ time). If the whole P' was matched, it did not contain the wildcard symbol and all the terminals in the subtree of v can be reported. If the first unmatched character of P' was not the wildcard, there are no occurrences. Otherwise, we need to select a character on an edge outgoing from v to be matched against the wildcard. First, we consider choosing the symbol on the heavy path of v. To this end, we ask an unrooted TreeLCP query on the node located one symbol below v on the heavy path and the suffix P'' of P starting immediately after the last wildcard symbol.

Next, we consider choosing the symbol that is not on the heavy path of v. In other words, we need to identify all trimmed suffixes among $S'_{v,1}, \ldots, S'_{v,b_v}$ that have P'' as a prefix. The result will consist of a connected sublist of this list that is computed in $\mathcal{O}(\log\log n)$ time using Lemma 4.5 for $k = k' = 0$. Here we use the fact that for any index i, we can identify the suffix $S'_{v,i}$ by computing $label'_v[i]$ in $\mathcal{O}(1)$ time by Lemma 7.5. To compare a suffix P' of P without wildcards with a suffix of T in $\mathcal{O}(1)$ time, we use Lemma 2.4 together with the data structure for LCP-queries in T [BF00].[4]

Overall, the query complexity is $\mathcal{O}(2^k \log\log n + m + \mathsf{occ})$, as desired. □

7.2 k-Mismatch Indexing

We show how our approach from Section 7.1 can be used for k-Mismatch Indexing. Let us recall some parts of the approach of Chan et al. [CLS+10].[5]

7.2.1 Storing 1-modified suffixes at level k compactly

As before, we explicitly construct only the $(k-1)$-errata tree. Let $\mathcal{T}$ be a compact trie at level $k-1$. For a heavy path C in $\mathcal{T}$, let $\mathcal{T}_C$ be the subtree of the root of C. We consider all terminals in $\mathcal{T}_C$; for a terminal representing suffix S, we find the node v of C where the path representing S diverges from C (if any) and create a 1-modified suffix by changing the character after v in S to the character that follows v in the heavy path. Let $S'_{C,1}, \ldots, S'_{C,b_C}$ be the list of all such 1-modified suffixes, ordered lexicographically, and let $S_{C,1}, \ldots, S_{C,b_C}$ be the list of original suffixes but in the order of the 1-modified suffixes. Each terminal ℓ of $\mathcal{T}$ implies a 1-modified suffix $S'_{C,i}$ for $\mathcal{O}(\log n)$ heavy paths C on the root-to-ℓ path, so the total number of modified suffixes $S'_{C,i}$ is $\mathcal{O}(n \log^k n)$. Instead of storing modified suffixes $S'_{C,i}$ explicitly, we store in a compact way the indices $label_C[i] := n - 1 - |S'_{C,i}|$.

The next lemma is proved by representing in a compact way arrays *st-rank*, *tree-pointer*, and *modified-rank* defined for heavy paths of each compact trie, as well as a sparse suffix array for each compact trie.

Lemma 7.6 ([CLS+10, Section 3.3 and Lemma 11])**.** *There is a data structure of size $\mathcal{O}(n \log^{k-1} n)$ that, given a heavy path C of a compact trie $\mathcal{T}$ at level $k-1$ of the errata tree and index $i \in [1\,..\,b_C]$, returns $label_C[i]$ in $\mathcal{O}(1)$ time.*

[4] We could also use Lemma 3.3 but it is not necessary, as there are no modifications in suffixes.

[5] Actually, their approach was stated explicitly only for the case of $k = 1$; we make the necessary adaptations for an arbitrary k.

7.2.2 Compact k-errata tree

The main differences between the proof of the theorem below and of Theorem 1.3 are that: (1) here we use compact representations of both types of labels, $label_C$ for heavy paths C and $label'_v$ for nodes v, and (2) for reporting 1-mismatch occurrences using labels $label'_v$, we use an additional data structure to skip over potential exact occurrences to avoid reporting such occurrences multiple times. The data structure consists of bit arrays B together with queries: $rank_q(x) = |\{i \in [1\,..\,x] : B[i] = q\}|$, $select_q(i) = \min\{x \in [1\,..\,|B|] : rank_q(x) = i\}$, for $q \in \{0, 1\}$. It is known [Jac89] that such queries can be answered in $\mathcal{O}(1)$ time using a data structure of $o(|B|)$ bits.

Theorem 1.1. *For every text of length n and integer threshold $k \geq 1$, there exists a k-mismatch index of size $\mathcal{O}(n \log^{k-1} n)$ that answers queries in $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$ time, where m is the length of the pattern and occ is the number of k-mismatch occurrences.*

Proof. Data structure. We store the $(k-1)$-errata tree explicitly. We also store the data structure of Lemma 7.6 for computing $label_C$. Finally, we store the arrays $st\text{-}rank'_v$, $tree\text{-}pointer'_v$, and $modified\text{-}rank'_{v,u_a}$ with rank queries, as well as sparse suffix arrays for all compact tries at level $k-1$ as in wildcard indexing. The data structures take $\mathcal{O}(n \log^{k-1} n)$ space.

For each heavy path C of each compact trie $\mathcal{T}$ at level $k-1$ of the data structure, let $\mathcal{T}'_C$ denote the compact trie of 1-modified suffixes $S'_{C,1}, \ldots, S'_{C,b_C}$. We select every $\lfloor \log n \rfloor$th modified suffix among $S'_{C,1}, \ldots, S'_{C,b_C}$, as well as $S'_{C,1}$ and S'_{C,b_C}, and construct a compact trie $\mathcal{T}''_C$ of these modified suffixes. Each terminal $S'_{C,i}$ in $\mathcal{T}''_C$ stores i as its number. The total size of the compact tries $\mathcal{T}''_C$ is $\mathcal{O}(n \log^{k-1} n)$, whereas trees $\mathcal{T}'_C$ are *not* stored explicitly. We construct our data structure for answering unrooted $\mathsf{TreeLCP}$ queries on each of $\mathcal{T}''_C$, with just $k = 1$ modification (Lemma 3.2).

We also store the (analogous) compact tries $\mathcal{T}''_v$ of sampled terminals $S'_{v,1}$ with unrooted $\mathsf{TreeLCP}$ data structure exactly as in the proof of Theorem 1.3. A data structure for LCP-queries in T is stored [BF00]. Also, a data structure for kangaroo jumps (for $k = 1$) of Lemma 3.3 is stored.

Let v be an explicit node in a compact trie at level $k-1$. For $i \in [1\,..\,b_v]$, by $c_{v,i}$ we denote the first character of suffix $S_{v,i}$ (i.e., $S_{v,i} = c_{v,i} S'_{v,i}$). We store a bit array diff_v of size $b_v - 1$ such that $\mathit{diff}_v[i] = 1$ if and only if $c_{v,i} \neq c_{v,i+1}$, together with a data structure for $rank_q$ and $select_q$ queries [Jac89] on diff_v. The bitmasks and rank/select data structures take $\mathcal{O}(n \log^k n)$ bits in total, i.e., $\mathcal{O}(n \log^{k-1} n)$ words of space.

Queries. We execute a $(k-1)$-errata tree query for P. As a result, in time $\mathcal{O}(\log^{k-1} \log\log n)$, $\mathcal{O}(\log^{k-1} n)$ pairs $(\mathcal{T}, P')$ are computed where $\mathcal{T}$ is a compact trie at level $k-1$ and P' is a suffix of P; our goal is to match P' against terminals in $\mathcal{T}$ with up to 1 mismatch. We compute $v = \mathsf{TreeLCP}(\mathcal{T}, P')$ (in $\mathcal{O}(\log\log n)$ time). If the whole P' was matched, all the terminals in the subtree of v can be reported for the start.

Let us decompose the path from the root of $\mathcal{T}$ to v into $h = \mathcal{O}(\log n)$ prefix fragments $C'_1, \ldots, C'_h$ of heavy paths $C_1, \ldots, C_h$, top-down. First, we find 1-mismatch matches of P' that contain the mismatch within one of $C'_1, \ldots, C'_h$, say C'_p. In other words, we need to identify all 1-modified suffixes among $S'_{C_p,1}, \ldots, S'_{C_p,b_{C_p}}$ that have a certain suffix P'' of P' as a prefix. The result will consist of a connected sublist of this list that is computed in $\mathcal{O}(\log\log n)$ time using Lemma 4.5 for $k = 1$ and $k' = 0$, with the aid of queries to $label_{C_p}$ array (Lemma 7.6) and kangaroo jumps (Lemma 3.3).

We find the remaining 1-mismatch matches of P' similarly as in Theorem 1.3, but using the *diff* arrays. If such a match contains the character that immediately follows C'_p on heavy path C_p, it is found using an unrooted $\mathsf{TreeLCP}$ query on $\mathcal{T}$. Otherwise, let v_p be the bottommost node of C'_p. First let $p < h$. Then the result will correspond to a connected sublist of $S'_{v_p,1}, \ldots, S'_{v_p,b_{v_p}}$ but

without the elements i such that $c_{v_p,i}$ matches the first character of P' after having matched C'_p (to avoid reporting exact matches of P' multiple times). We find the connected sublist in $\mathcal{O}(\log\log n)$ time using Lemma 4.5 for $k = k' = 0$ and a suffix of P', with the aid of queries to label'_{v_p} array. Then we report only the trimmed suffixes for which the preceding character is not equal to the corresponding character of P' using rank/select on diff_{v_p} array. More precisely, whenever we find a trimmed suffix $S'_{v_p,i}$ that does have the unwanted preceding character $c_{v_p,i}$, we skip to the next one that does not by identifying the nearest index $i' \geq i$ such that $\mathit{diff}_{v_p}[i'] = 1$. Thus, at most every second step a new occurrence is reported. If $p = h$, we do not need to use the diff_{v_p} array as P' was not matched beyond C'_h.

After $\mathcal{O}(m)$ preprocessing, for each initial pair $(\mathcal{T}, P')$, we consider $\mathcal{O}(\log n)$ heavy paths and for each of them perform computations in $\mathcal{O}(\log\log n)$ time, plus the output size. Overall, the query complexity is $\mathcal{O}(\log^k n \log\log n + m + \mathsf{occ})$, as desired. □

8 Proofs of Lemmas from Section 3

In the weighted ancestor queries problem over compact trie $\mathcal{T}$, given a node v of $\mathcal{T}$ and an integer $\ell \geq 0$, we are to compute the (explicit or implicit) node w that is an ancestor of v and has string depth ℓ. For a compact trie $\mathcal{T}$, there is a data structure of $\mathcal{O}(|\mathcal{T}|)$ size that can answer weighted ancestor queries on $\mathcal{T}$ in $\mathcal{O}(\log\log|\mathcal{T}|)$ time [ALLS07].

Lemma 3.1. *Theorem 2.3 holds also if $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are compact tries of fragments of T, each given in a canonical form.*

Proof. For each tree $\mathcal{T}_i$, $i \in [1\,..\,t]$, we create a compact trie $\mathcal{T}'_i$ of suffixes of T as follows: for every fragment $T[a\,..\,b)$ that is a terminal in $\mathcal{T}_i$, suffix $T[a\,..\,n)$ is represented in $\mathcal{T}'_i$. Each suffix represented in $\mathcal{T}'_i$ stores one fragment from $\mathcal{T}_i$ it originates from. Each explicit node of each $\mathcal{T}'_i$ stores an arbitrary terminal from its subtree. Moreover, each explicit node of each $\mathcal{T}'_i$ stores the corresponding node—that is, the node with the same path label—in $\mathcal{T}_i$, if it exists, and *vice versa*. We create the data structure of Theorem 2.3 for answering unrooted $\mathsf{TreeLCP}$ queries in tries $\mathcal{T}'_1, \ldots, \mathcal{T}'_t$. For each of the tries $\mathcal{T}'_i$, we also store a data structure for weighted ancestor queries on $\mathcal{T}'_i$ [ALLS07].

Let v' be the node of $\mathcal{T}'_i$ that corresponds to v in $\mathcal{T}_i$. Node v' is computed via the nearest explicit descendant y of v, its corresponding node y' in $\mathcal{T}'_i$, and a weighted ancestor query from y'. Upon a query $\mathsf{TreeLCP}_v(\mathcal{T}_i, P')$, we ask a query $\mathsf{TreeLCP}_{v'}(\mathcal{T}'_i, P')$. Let w' be the resulting node, $T[a\,..\,n)$ be the string label of any terminal in the subtree of w', and $T[a\,..\,b)$ be the corresponding fragment of T that was represented in $\mathcal{T}_i$. We ask a weighted ancestor query for w' at depth $\min(b-a, d)$, where d is the string depth of w', that returns a node u'. Then $\mathsf{TreeLCP}_v(\mathcal{T}_i, P')$ returns the node u in $\mathcal{T}_i$ that corresponds to u'.

Let us argue for the correctness of the query procedure. Let node x in $\mathcal{T}_i$ be the correct result of $\mathsf{TreeLCP}_v(\mathcal{T}_i, P')$ query. The path label of u' is a prefix of $T[a\,..\,b)$, so node u indeed exists in $\mathcal{T}_i$. We will show that $x = u$. The path from v' to w' has a label being a prefix of P'. Hence, the path from v' to u' (equivalently, the path from v to u) has a label $P'[0\,..\,z)$ for some index z. This concludes that x is a descendant of u. Assume to the contrary that x is a strict descendant of u. In this case, $u' \neq w'$ and u' is a terminal with path label $T[a\,..\,b)$. Then the label of the path from v to x has a prefix $P'[0\,..\,z]$. Let X be the path label of v. String $X \cdot P'[0\,..\,z]$ is a prefix of $T[a\,..\,n)$, so it equals $T[a\,..\,b]$. As x in present in $\mathcal{T}_i$, this means that $\mathcal{T}_i$ is not given in a canonical form, as the terminal representing $T[a\,..\,b)$ has an outgoing edge starting with label $T[b]$; a contradiction.

All data structures use $\mathcal{O}(n + \sum_{i=1}^{t}(|\mathcal{T}_i| + |\mathcal{T}'_i|)) = \mathcal{O}(n + \sum_{i=1}^{t}|\mathcal{T}_i|)$ space and any unrooted query $\mathsf{TreeLCP}_v(\mathcal{T}_i, P)$ is answered in $\mathcal{O}(\log\log n)$ time (after $\mathcal{O}(m)$ preprocessing for pattern P, $|P| = m$, that is required in Theorem 2.3). □

Lemma 3.2. *For all constants $k, k' = \mathcal{O}(1)$, Theorem 2.3 holds also if $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are compact tries of ($\leq k$)-modified suffixes of T and queries are for ($\leq k'$)-modified suffixes of P.*

Proof. Data structure. Each trie $\mathcal{T}_i$, for $i \in [1\,..\,t]$, is (edge-)decomposed into compact tries of factors of T and compact tries of single-character strings recursively in phases numbered 1,2,...; see Figure 2. Odd and even phases are processed differently. In an odd phase, assume a subtree $\mathcal{T}$ of $\mathcal{T}_i$ is still to be decomposed. For each ($\leq k$)-modified suffix of T that is present in $\mathcal{T}$, we select its longest prefix until the next modification. A compact trie of these prefixes, denoted $\mathcal{T}'$, forms a tree in the decomposition. The subtrees of $\mathcal{T}$ obtained upon the removal of $\mathcal{T}'$ become the input of an even phase. In an even phase, for each remaining tree, we form a tree of string depth one by taking the first character of every edge going from the root; the remaining trees are input to the next odd phase. The process ends when there are no further trees to be decomposed. Naturally, each tree in the decomposition stores links to the trees formed at a subsequent phase in the appropriate nodes (that were explicit in $\mathcal{T}$).

aababaaba
aababacb
abaaaaa
abaabb
bcab
bcba
bccaaa
bccabc

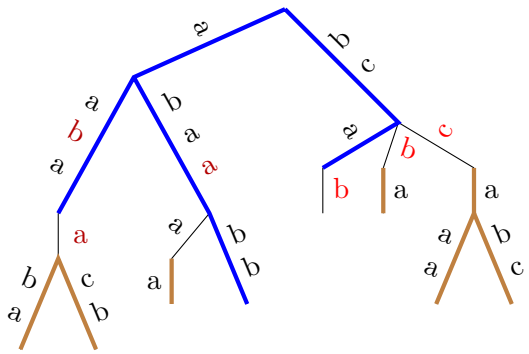

Figure 2: Left: an ordered list of (≤ 1)-modified fragments of some hypothetical text T; modifications are shown in red. (One of the fragments is actually 0-modified.) Right: the compact trie of these fragments decomposed as in the proof of Lemma 3.2. The subtrees in bold (blue and brown) are created in odd phases and are compact tries of substrings of T; the remaining even-phase subtrees have string depth 1. The characters drawn in half red, half black were modified in one fragment but unmodified in some other one.

Each compact trie created in an odd phase represents a set of fragments of T. As described in Section 3, it can be represented in a canonical form by extending the terminals. Thus, a data structure of Lemma 3.1 for all trees in the decomposition created in odd phases can be stored. The suffix tree of T is also stored. Each explicit node of a tree $\mathcal{T}'$ of the decomposition stores the corresponding explicit node of $\mathcal{T}_i$, and each explicit node of $\mathcal{T}_i$ stores the corresponding node(s) of trees in the decomposition (there are up to two such trees, one from an odd phase and one from an even phase). We also store the data structure of Theorem 2.2 for each tree in the decomposition.

Claim 8.1. If $k \geq 1$, the total size of trees in the decomposition is $\mathcal{O}(Nk)$ and the number of phases is $\mathcal{O}(k)$.

Proof. Let S be a k''-modified suffix of T being a terminal of $\mathcal{T}_i$. It suffices to show that fragments originating from S are present in at most $2k''$ subsequent trees in the decomposition of $\mathcal{T}_i$. If $k'' = 0$,

the whole S is contained already in the topmost tree. Assume now that $k'' > 0$.

Let $\mathcal{T}$ be the topmost tree in the decomposition of $\mathcal{T}_i$ and a be the position of the first modification in S. By definition, $\mathcal{T}$ contains a prefix of S of length at least a which is trimmed for the next phase. The next tree, created at an even phase, removes one character from the remaining suffix. Effectively, by the next odd phase, a prefix of S containing the first modification is removed, and we are left with a $(k''-1)$-modified suffix of S. The conclusion follows by induction. □

Preprocessing of the pattern. We perform the preprocessing of Lemma 3.1 for odd-phase trees that takes $\mathcal{O}(m)$ time.

Query. Let us consider a query $\mathsf{TreeLCP}_v(\mathcal{T}_i, P')$ for v being a node of $\mathcal{T}_i$ and P' being a $(\leq k')$-modified suffix of the pattern. Then P' can be partitioned into $\mathcal{O}(k'+1)$ fragments, each of which is a fragment of P or a single character originating from a modification. The fragments in the partition are processed consecutively. At each point, a node u in a tree $\mathcal{T}$ from the decomposition of $\mathcal{T}_i$ is stored; the node is such that the path label of the corresponding node u' in $\mathcal{T}_i$ equals the already processed prefix of P'. Initially, u and $\mathcal{T}$ are such that u corresponds to the node v in $\mathcal{T}_i$.

A fragment F in the partition of P' is processed as follows. First, we ask a $\mathsf{TreeLCP}_u(\mathcal{T}, F)$ query and obtain a node w in $\mathcal{T}$ that corresponds to a node w' of $\mathcal{T}_i$. If $\mathcal{T}$ was formed at an even phase, this query is simply processed in $\mathcal{O}(1)$ time. If $|F| = 1$, we apply Theorem 2.2 in $\mathcal{O}(1)$ time. Otherwise, the query is answered in $\mathcal{O}(\log\log n)$ time by Lemma 3.1. If the length ℓ of the path from u to w is smaller than $|F|$, a prefix of F of length ℓ is removed, and the next call will be for the root of the next tree in the decomposition that starts in w, if there is any such tree, and otherwise, node w' of $\mathcal{T}_i$ is returned. Finally, if $\ell = |F|$, the next call will be for node w in $\mathcal{T}$ and the next fragment of P'.

Each subsequent call removes one of the $\mathcal{O}(k'+1)$ fragments of P' or moves to the next tree in the decomposition. By the claim, the query complexity is $\mathcal{O}((k+k'+1)\log\log n)$. □

Lemma 3.3. *A length-n text T can be enhanced with a data structure of size $\mathcal{O}(n)$ that, given a length-m pattern P, preprocesses P in $\mathcal{O}(m)$ time so that later, given a k-modified suffix S of T and a k'-modified suffix P' of P, we can compute $\mathsf{LCP}(S, P')$ in $\mathcal{O}(k+k'+1)$ time.*

Proof. Data structure. We use the suffix tree $\widetilde{\mathcal{T}}$ of T and the data structure [BF00] for LCP-queries on T. We also use the $\mathcal{O}(n)$-space data structure of Lemma 2.4.

Preprocessing of the pattern. We compute $\mathsf{TreeLCP}(\widetilde{\mathcal{T}}, U)$ for all suffixes U of pattern P in $\mathcal{O}(m)$ total time using Lemma 2.4.

Query. Let S and P' be as in the statement of the lemma. We partition S into $\mathcal{O}(k)$ fragments $X_1, \ldots, X_x$, each being a fragment of T or a single-character modification. Similarly, we partition P' into $\mathcal{O}(k')$ fragments $Y_1, \ldots, Y_y$, each being a fragment of P or a single-character modification. Each subsequent fragment Y_i such that $|Y_i| > 1$ is partitioned recursively by cutting off its longest prefix being a substring of T. That is, if the current Y_i is a prefix of a suffix U of P, we cut off the prefix of Y_i of length $\min(|Y_i|, \ell)$ where ℓ is the string depth of node $\mathsf{TreeLCP}(\widetilde{\mathcal{T}}, U)$. This process is interrupted if more than $y + 2x + 1$ fragments in the partition of P' have been formed; in this case, the remaining suffix V of P' is discarded. Thus, the partitioning takes $\mathcal{O}(k+k'+1)$ time. Let $Z_1 \cdots Z_z V$ be the final partition of P'.

For each Y_i such that $|Y_i| > 1$, all fragments formed from it except for the last one are called *special*.

Claim 8.2. Let X be a single-character string or a substring of T and $j \in [1\,..\,z)$ be such that Z_j is special. Denote $Z' = Z_j(Z_{j+1}[0])$ and $\ell = \mathsf{LCP}(X, Z')$. Then either $\ell = |X|$ or $\ell < \min(|X|, |Z'|)$.

Proof. If $|X| = 1$, then $\ell \in \{0, 1\}$ and the conclusion holds. Otherwise, X is a substring of T. If X is a prefix of Z', then $\ell = |X|$. By definition, string Z' is not a substring of T. Hence, Z' is not a prefix of X. If X is not a prefix of Z', then the longest common prefix of X and Z' is shorter than the two strings. □

```
1  S := X_1 ··· X_x;                                   // |X_i| = 1 or X_i is a fragment of T
2  P' := Z_1 ··· Z_z V;          // |Z_j| = 1 or Z_j is a fragment of T and V is a discarded suffix
3  i := j := 1;
4  lcp := 0;
5  while i ≤ x and j ≤ z do
6      if |X_i| = 1 then
7          if Z_j[0] ≠ X_i then break;
8          if |Z_j| = 1 then j := j + 1;
9          else Z_j := Z_j[1 .. |Z_j|);
10         i := i + 1;
11         lcp := lcp + 1;
12     else if |Z_j| = 1 then
13         if X_i[0] ≠ Z_j then break;
14         X_i := X_i[1 .. |X_i|);
15         j := j + 1;
16         lcp := lcp + 1;
17     else                                            // X_i and Z_j are fragments of T
18         ℓ := LCP(X_i, Z_j);
19         lcp := lcp + ℓ;
20         if ℓ < min(|X_i|, |Z_j|) then break;
21         if ℓ = |X_i| then i := i + 1;
22         else X_i := X_i[ℓ .. |X_i|);
23         if ℓ = |Z_j| then j := j + 1;
24         else Z_j := Z_j[ℓ .. |Z_j|);
25 return lcp;
```

Algorithm 1: Compute $\mathsf{LCP}(S, P')$

We compute $\mathsf{LCP}(S, P')$ by subsequently removing common prefixes of the two strings, as shown in Algorithm 1. Let X_i and Z_j be the fragments being the current prefixes of S and P', respectively. If $|X_i| = 1$ or $|Z_j| = 1$, we check if it agrees with the first character of the other string and either break the computations or cut the first character from both strings. Otherwise, we are left with computing the LCP of two substrings of T, which can be done in $\mathcal{O}(1)$ time [BF00].

Each step of the while-loop is performed in $\mathcal{O}(1)$ time and removes a fragment from the decomposition of S or P', or breaks the loop. By the claim, whenever a special fragment of P' followed by another fragment is processed, a whole fragment of S is removed or the algorithm breaks. In summary, at most $2x$ special fragments can be processed in the algorithm, which shows correctness of interrupting the partitioning of P'. The query algorithm works in $\mathcal{O}(k + k' + 1)$ time. □

9 Conclusions

We presented the first general space improvement—by a factor of $\Theta(\log n)$—for k-mismatch indexing over the errata trees of Cole, Gottlieb, and Lewenstein [CGL04], while retaining the same

query time for $k = \mathcal{O}(1)$. A variant of our index optimized for the case of $k, \sigma = \mathcal{O}(1)$ achieves an overall space improvement of nearly $\log^2 n$, surpassing the $\Theta(\log n)$-factor improvement of Chan, Lam, Sung, Tam, and Wong [CLS+10] for $\sigma = \mathcal{O}(1)$.

An immediate open problem is to design even more space-efficient k-mismatch indexes; our intermediate results imply that one can focus on patterns of polylogarithmic length. Another natural question is whether the query time of k-errata trees can be reduced.

The original k-errata tree can be constructed in time proportional to its size [CGL04] for $k = \mathcal{O}(1)$. We believe our data structures can be built in similar time, possibly with a small polylogarithmic-factor overhead. However, some of the black-box components we rely on, particularly Theorem 2.1 (from [RRR07]) and Theorem 2.3 (from [CLS+10]), lack efficient construction algorithms in the literature. Unpacking these components to provide efficient preprocessing is likely possible, though somewhat tedious.

We hope that the techniques developed in this work will prove useful in other applications of k-errata trees (see the references in Section 1). In ongoing work, we are exploring extensions to k-error text indexing.

References

[ABR00] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 198–207. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892088.

[AKL+00] Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000. doi:10.1006/JAGM.2000.1104.

[Al-15] Anas Al-Okaily. Error tree: A tree structure for Hamming and edit distances and wildcards matching. *J. Comput. Biol.*, 22(12):1118–1128, 2015. doi:10.1089/CMB.2015.0132.

[ALLS07] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

[ALP04] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.

[ALP25] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns using locally consistent anchors. *VLDB J.*, 34(5):58, 2025. doi:10.1007/S00778-025-00935-7.

[AN16] Peyman Afshani and Jesper Sindahl Nielsen. Data structure lower bounds for document indexing problems. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 93:1–93:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ICALP.2016.93.

[Bel09] Djamal Belazzougui. Faster and space-optimal edit distance "1" dictionary. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 154–167. Springer, 2009. doi:10.1007/978-3-642-02441-2_14.

[Bel15] Djamal Belazzougui. Improved space-time tradeoffs for approximate full-text indexing with one edit error. *Algorithmica*, 72(3):791–817, 2015. doi:10.1007/S00453-014-9873-9.

[BF00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

[BGP+24] Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error or mismatch. *Theory Comput. Syst.*, 68(5):1442–1467, 2024. doi:10.1007/S00224-024-10194-8.

[BGVV14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014. doi:10.1007/S00224-013-9498-4.

[BGW00] Adam L. Buchsbaum, Michael T. Goodrich, and Jeffery R. Westbrook. Range searching over tree cross products. In Mike Paterson, editor, *Algorithms - ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2000. doi:10.1007/3-540-45253-2_12.

[BII+17] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

[Boo80] Kellogg S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980. doi:10.1016/0020-0190(80)90149-0.

[CCI+18] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Linear-time algorithm for long LCF with k mismatches. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CPM.2018.23.

[CEK+21] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021. doi:10.1145/3426473.

[CFP+16] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In Robert Krauthgamer, editor,

Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, pages 2039–2052. SIAM, 2016. doi:10.1137/1.9781611974331.CH142.

[CFS19] Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1146–1164. SIAM, 2019. doi:10.1137/1.9781611975482.70.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.

[CIK+14] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/J.TCS.2013.11.018.

[CIK+22] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, and Juliusz Straszynski. Efficient computation of sequence mappability. *Algorithmica*, 84(5):1418–1440, 2022. doi:10.1007/S00453-022-00934-Y.

[CKP+21] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular pattern matching with k mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.

[CKPR21] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ESA.2021.30.

[CLS+10] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010. doi:10.1007/S00453-008-9263-2.

[CLS+11] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. *J. Discrete Algorithms*, 9(4):358–364, 2011. doi:10.1016/J.JDA.2011.04.004.

[CN21] Yangjun Chen and Hoang Hai Nguyen. On the string matching with k differences in DNA databases. *Proc. VLDB Endow.*, 14(6):903–915, 2021. doi:10.14778/3447689.3447695.

[CO06] Luís Pedro Coelho and Arlindo L. Oliveira. Dotted suffix trees. A structure for approximate text indexing. In Fabio Crestani, Paolo Ferragina, and Mark Sanderson, editors, *String Processing and Information Retrieval, 13th International Conference, SPIRE*

2006, Glasgow, UK, October 11-13, 2006, Proceedings, volume 4209 of *Lecture Notes in Computer Science*, pages 329–336. Springer, 2006. doi:10.1007/11880561_27.

[Cob95] Archie L. Cobbs. Fast approximate matching using suffix trees. In Zvi Galil and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, July 5-7, 1995, Proceedings*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1995. doi:10.1007/3-540-60044-2_33.

[CW18] Yangjun Chen and Yujia Wu. BWT: An index structure to speed-up both exact and inexact string matching. In Sanjiban Sekhar Roy, Pijush Samui, Ravinesh Deo, and Stavros Ntalampiras, editors, *Big Data in Engineering Applications*, pages 221–264, Singapore, 2018. Springer Singapore. doi:10.1007/978-981-10-8476-8_12.

[Duv83] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

[EF21] Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.63.

[EGM+07] Chiara Epifanio, Alessandra Gabriele, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Languages with mismatches. *Theor. Comput. Sci.*, 385(1-3):152–166, 2007. doi:10.1016/J.TCS.2007.06.006.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

[FG15] Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015. doi:10.1007/978-3-319-19929-0_14.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

[GG87] Zvi Galil and Raffaele Giancarlo. Parallel string matching with k mismatches. *Theor. Comput. Sci.*, 51:341–348, 1987. doi:10.1016/0304-3975(87)90042-9.

[GGM+25] Pawel Gawrychowski, Adam Górkiewicz, Pola Marciniak, Solon P. Pissis, and Karol Pokorski. Faster approximate elastic-degenerate string matching - part B. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025*, volume 331 of *LIPIcs*, pages 29:1–29:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.CPM.2025.29.

[GHPT25] Daniel Gibney, Jackson Huffstutler, Mano Prakash Parthasarathi, and Sharma V. Thankachan. Repetition aware text indexing for matching patterns with wildcards. In Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis, editors, *52nd International Colloquium on Automata, Languages, and Programming, ICALP 2025*, volume 334 of *LIPIcs*, pages 88:1–88:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ICALP.2025.88.

[GLS18] Pawel Gawrychowski, Gad M. Landau, and Tatiana Starikovskaya. Fast entropy-bounded string dictionary look-up with mismatches. In Igor Potapov, Paul G. Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018*, volume 117 of *LIPIcs*, pages 66:1–66:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.MFCS.2018.66.

[GMRS03] Alessandra Gabriele, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Indexing structures for approximate string matching. In Rossella Petreschi, Giuseppe Persiano, and Riccardo Silvestri, editors, *Algorithms and Complexity, 5th Italian Conference, CIAC 2003, Rome, Italy, May 28-30, 2003, Proceedings*, volume 2653 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2003. doi:10.1007/3-540-44849-7_20.

[GNP20] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

[GU18] Pawel Gawrychowski and Przemyslaw Uznanski. Towards unified approximate pattern matching for Hamming and L_1 distance. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 62:1–62:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.ICALP.2018.62.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

[HHLS04] Trinh N. D. Huynh, Wing-Kai Hon, Tak Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. In Süleyman Cenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusöz, editors, *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, volume 3109 of *Lecture Notes in Computer Science*, pages 434–444. Springer, 2004. doi:10.1007/978-3-540-27801-6_33.

[HKS+13] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressed text indexing with wildcards. *J. Discrete Algorithms*, 19:23–29, 2013. doi:10.1016/J.JDA.2012.12.003.

[HLS+11] Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Cache-oblivious index for approximate string matching. *Theor. Comput. Sci.*, 412(29):3579–3588, 2011. doi:10.1016/J.TCS.2011.03.004.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

[KK99] Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

[KK19] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

[KK21] Dominik Kempa and Tomasz Kociumaka. Breaking the O(n)-barrier in the construction of compressed suffix arrays. *CoRR*, abs/2106.12725, 2021. arXiv:2106.12725.

[KK23] Dominik Kempa and Tomasz Kociumaka. Breaking the O(n)-barrier in the construction of compressed suffix arrays and suffix trees. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 5122–5202. SIAM, 2023. doi:10.1137/1.9781611977554.CH187.

[KNO24] Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86(4):1031–1056, 2024. doi:10.1007/S00453-023-01186-0.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

[KST16] Gregory Kucherov, Kamil Salikhov, and Dekel Tsur. Approximate string matching using a bidirectional index. *Theor. Comput. Sci.*, 638:145–158, 2016. doi:10.1016/J.TCS.2015.10.043.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinform.*, 25(14):1754–1760, 2009. doi:10.1093/BIOINFORMATICS/BTP324.

[Lew16] Moshe Lewenstein. Dictionary matching. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 533–538. Springer New York, 2016. doi:10.1007/978-1-4939-2864-4_109.

[LLT+09] Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36. IEEE Computer Society, 2009. doi:10.1109/BIBM.2009.42.

[LMRT14] Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, and Sharma V. Thankachan. Less space: Indexing for queries with wildcards. *Theor. Comput. Sci.*, 557:120–127, 2014. doi:10.1016/J.TCS.2014.09.003.

[LNV14] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 506–517. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPICS.STACS.2014.506.

[Lot97] M. Lothaire. *Combinatorics on words, Second Edition*. Cambridge mathematical library. Cambridge University Press, 1997.

[LSTY07] Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with don't cares. In Takeshi Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 846–857. Springer, 2007. doi:10.1007/978-3-540-77120-3_73.

[LSW08] Tak Wah Lam, Wing-Kin Sung, and Swee-Seong Wong. Improved approximate string matching using compressed suffix data structures. *Algorithmica*, 51(3):298–314, 2008. doi:10.1007/S00453-007-9104-8.

[LV86] Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.

[LV89] Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989. doi:10.1016/0196-6774(89)90010-2.

[Maa06] Moritz G. Maaß. Average-case analysis of approximate trie search. *Algorithmica*, 46(3-4):469–491, 2006. doi:10.1007/S00453-006-0126-4.

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

[MN05] Moritz G. Maaß and Johannes Nowak. A new method for approximate indexing and dictionary lookup with one error. *Inf. Process. Lett.*, 96(5):185–191, 2005. doi:10.1016/J.IPL.2005.08.001.

[MN07] Moritz G. Maaß and Johannes Nowak. Text indexing with errors. *J. Discrete Algorithms*, 5(4):662–681, 2007. doi:10.1016/J.JDA.2006.11.001.

[Mun96] J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996. doi:10.1007/3-540-62034-6_35.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

[Nav22] Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2022. doi:10.1145/3432999.

[RI07] M. Sohel Rahman and Costas S. Iliopoulos. Pattern matching algorithms with don't cares. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, Frantisek Plásil, and Mária Bieliková, editors, *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings Volume II*, pages 116–126. Institute of Computer Science AS CR, Prague, 2007.

[RRR07] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

[TAA16] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *J. Comput. Biol.*, 23(6):472–482, 2016. doi:10.1089/CMB.2015.0235.

[Tha13] Chris Thachuk. Compressed indexes for text with wildcards. *Theor. Comput. Sci.*, 483:22–35, 2013. doi:10.1016/J.TCS.2012.08.011.

[Tsu10] Dekel Tsur. Fast index for approximate string matching. *J. Discrete Algorithms*, 8(4):339–345, 2010. doi:10.1016/J.JDA.2010.08.002.

[TWLY09] Alan Tam, Edward Wu, Tak Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, volume 5721 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2009. doi:10.1007/978-3-642-03784-9_5.

[Ukk93] Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93, Padova, Italy, June 2-4, 1993, Proceedings*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1993. doi:10.1007/BFB0029808.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

[ZLPT24] Wiktor Zuba, Grigorios Loukides, Solon P. Pissis, and Sharma V. Thankachan. Approximate suffix-prefix dictionary queries. In Rastislav Královic and Antonín Kucera, editors, *49th International Symposium on Mathematical Foundations of Computer Science, MFCS 2024*, volume 306 of *LIPIcs*, pages 85:1–85:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.MFCS.2024.85.