Streaming Dictionary Matching with Mismatches

Paweł Gawrychowski
University of Wrocław, 50-137 Wrocław, Poland
gawry@cs.uni.wroc.pl

Tatiana Starikovskaya
DIENS, École normale supérieure, PSL Research University, 75005 Paris, France
tat.starikovskaya@gmail.com

Abstract

In the k-mismatch problem we are given a pattern of length m and a text and must find all locations where the Hamming distance between the pattern and the text is at most k. A series of recent breakthroughs have resulted in an ultra-efficient streaming algorithm for this problem that requires only $\mathcal{O}(k \log \frac{m}{k})$ space [Clifford, Kociumaka, Porat, SODA 2019]. In this work, we consider a strictly harder problem called dictionary matching with k mismatches, where we are given a dictionary of d patterns of lengths $\leq m$ and must find all their k-mismatch occurrences in the text, and show the first streaming algorithm for it. The algorithm uses $\mathcal{O}(kd \log^k d \operatorname{polylog} m)$ space and processes each position of the text in $\mathcal{O}(k \log^k d \operatorname{polylog} m + occ)$ time, where occ is the number of k-mismatch occurrences of the patterns that end at this position. The algorithm is randomised and outputs correct answers with high probability.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Streaming, multiple pattern matching, Hamming distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.21

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving this paper.

1 Introduction

The pattern matching problem is the fundamental problem of string processing and has been studied for more than 40 years. Most of the existing algorithms are deterministic and assume the word-RAM model of computation. Under these assumptions, we must store the input in full, which is infeasible for modern massive data applications. The streaming model of computation was designed to overcome the restrictions of the word-RAM model. In this model, we assume that the text arrives as a stream, one character at a time. The characters are assumed to be integers that fit in $\mathcal{O}(\log n)$-bit machine words, where n is the length of the stream. Each time a new character of the text arrives, we must update the output. The space complexity of an algorithm is defined to be all the space used, including the space we need to store the information about the pattern(s) and the text. The time complexity of an algorithm is defined to be the time we spend to process one character of the text. The streaming model of computation aims for algorithms that use as little space and time as possible. All streaming algorithms we discuss in this paper are randomised by necessity. They can err with probability inverse-polynomial in the length of the input.

The first sublinear-space streaming algorithm for exact pattern matching was suggested by Porat and Porat in FOCS 2009 [26]. For a pattern of length m, their algorithm uses $\mathcal{O}(\log m)$ space and $\mathcal{O}(\log m)$ time per character. Later, Breslauer and Galil gave a $\mathcal{O}(\log m)$-space and $\mathcal{O}(1)$-time algorithm [8].

The first algorithm for dictionary matching was developed by Aho and Corasick [1]. The algorithm assumes the word-RAM model of computation, and for a dictionary of d patterns of length at most m, uses $\Omega(md)$ space and $\mathcal{O}(1 + occ)$ amortised time per

© Paweł Gawrychowski and Tatiana Starikovskaya;
licensed under Creative Commons License CC-BY
30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis; Article No. 21; pp. 21:1–21:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

character, where *occ* is the number of the occurrences ending at this position. Apart from the Aho–Corasick algorithm, other word-RAM algorithms for exact dictionary matching include [3, 4, 7, 13, 14, 16, 19, 21, 22, 27]. In ESA 2015, Clifford et al. [9] showed a streaming dictionary matching algorithm that uses $\mathcal{O}(d \log m)$ space and $\mathcal{O}(\log\log(m+d) + occ)$ time per character. In ESA 2017, Golan and Porat [18] showed an improved algorithm that uses the same amount of space and $\mathcal{O}(1 + occ)$ time per character for constant-size alphabets.

In the k-mismatch problem we are given a pattern of length m and a text and must find all alignments of the pattern and the text where the Hamming distance is at most k. By reduction to the streaming exact pattern matching, Porat and Porat [26] showed the first streaming k-mismatch algorithm with space $\mathcal{O}(k^3 \log^7 m/\log\log m)$ and time $\mathcal{O}(k^2 \log^5 m/\log\log m)$. The complexity has been subsequently improved in [10, 11, 17]. The current best algorithm uses only $\mathcal{O}(k \log \frac{m}{k})$ space and $\mathcal{O}(\log \frac{m}{k}(\sqrt{k \log k} + \log^3 m))$ time per character [11].

In the problem of dictionary matching with k mismatches, we are given a set (dictionary) of d patterns of maximal length m and must find all their k-mismatch occurrences in the text. This problem is strictly harder than both k-mismatch and dictionary matching, and on the other hand, it is well-motivated by practical applications in cybersecurity and bioinformatics. In the word-RAM model, dictionary matching with k mismatches has been addressed in [2, 24, 25]. Muth and Manber [24] gave a randomised algorithm for $k = 1$, and Baeza-Yates and Navarro [2] and Navarro [25] gave the first algorithms for a general value of k. The time complexity of the algorithms is good on average, but in the worst case can be $\Omega(md)$ per character.

1.1 Our results

In this work, we commence a study of dictionary matching with k mismatches in the streaming model of computation. Our contribution is twofold. First, we show a streaming dictionary matching algorithm that uses space sublinear in m and time sublinear in both m and d (Section 4). Similar to previous work on streaming pattern matching, we assume that we receive the dictionary first, preprocess it (without accounting for the preprocessing time), and then receive the text.

▶ **Theorem 1.** *For any $k \geq 1$, there is a streaming algorithm that solves dictionary matching with k mismatches in $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ worst-case time per arriving character. The algorithm is randomised and its answers are correct w.h.p.*[1] *Both false-positive and false-negative errors are allowed.*

Hereafter *occ* is the number of k-mismatch occurrences of the patterns that end at the currently processed position of the text, i.e., it is at most d and typically it is much smaller than the total number of the occurrences of the patterns in the text. Our algorithm makes use of a new randomised variant of the k-errata tree (Section 3), a famous data structure of Cole, Gottlieb, and Lewenstein for dictionary matching with k mismatches [12]. This variant of the k-errata tree allows to improve both the query time and the space requirements and can be considered as a generalisation of the z-fast tries [5, 6], that have proved to be useful in many streaming applications.

Compare our result to a streaming algorithm that can be obtained by a repeated application of the k-mismatch algorithm [11]:

[1] $\tilde{\mathcal{O}}$ hides a multiplicative factor polynomial in $\log m$ and w.h.p. means that the error probability is at most $1/n^c$ for an arbitrary given constant c.

▶ **Corollary 2** (of Clifford, Kociumaka, and Porat [11]). *For any $k \geq 1$, there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(dk)$ space and $\tilde{\mathcal{O}}(d\sqrt{k})$ time per character. The algorithm is randomised and its answers are correct w.h.p.*

As it can be seen, the time complexity of Corollary 2 depends on d in linear way, which is prohibitive for applications where the stream characters arrive at a high speed and the size of the dictionary is large, up to several thousands of patterns, as we must be able to process each character before the next one arrives to benefit from the space advantages of streaming algorithms.

Our second contribution is a space lower bound for streaming dictionary matching with mismatches. In Section 6 we show the following claim by reduction from the Index problem (see the proof for the definition):

▶ **Lemma 3.** *Any streaming algorithm for dictionary matching with k mismatches such that its answers are correct w.h.p. requires $\Omega(kd)$ bits of space.*

2 Preliminaries: Fingerprints and sketches

In this section, we give the definitions of two hash functions that we use throughout the paper. We first give the definition of Karp–Rabin fingerprints that let us decide whether two strings are equal.

▶ **Definition 4** (Karp–Rabin fingerprints, Karp–Rabin [20]). *For a fixed prime p and $r \in [0, p-1]$ chosen uniformly at random, the Karp—Rabin fingerprint of a string $S = S[1]S[2]\dots S[m]$ is defined as a quadruple $\Phi(S) = (\varphi(S), \varphi^R(S), r^{|S|} \bmod p, r^{-|S|} \bmod p)$, where $\varphi(S) = \sum_{i=1}^{m} S[i] \cdot r^{m-i} \bmod p$ and $\varphi^R(S) = \sum_{i=1}^{m} S[i] \cdot r^{i-1} \bmod p$.*

▶ **Fact 5.** *For $r \in [0, p-1]$ chosen uniformly at random, the probability of two distinct strings of equal lengths $\ell \leq m$ over the integer alphabet $[0, p-1]$ to have equal Karp–Rabin fingerprints is at most m/p.*

Consider a string Z that is equal to the concatenation of two strings X and Y of length at most m, that is $Z = XY$. We can compute in $\mathcal{O}(1)$ time $\varphi(Z)$ given $\varphi(X)$ and $\varphi(Y)$, and $\varphi(Y)$ given $\varphi(Z)$ and $\varphi(X)$. Furthermore, given the Karp–Rabin fingerprint of $S[1]S[2]\dots S[m]$, we can compute the Karp–Rabin fingerprint of $S[m]S[m-1]\dots S[1]$ in $\mathcal{O}(1)$ time.

We now remind the definition of k-mismatch sketches that will allow us to decide whether two strings are at Hamming distance at most k.

▶ **Definition 6** (k-mismatch sketch, Clifford, Kociumaka, and Porat [11]). *For a fixed prime p and $r \in [0, p-1]$ chosen uniformly at random, the k-mismatch sketch $sk_k(S)$ of a string $S = S[1]S[2]\dots S[m]$ is defined as a tuple $(\phi_0(S), \dots, \phi_{2k}(S), \phi'_0(S), \dots, \phi'_k(S), \Phi(S))$, where $\phi_j(S) = \sum_{i=1}^{i=m} S[i] \cdot i^j \bmod p$ and $\phi'_j(S) = \sum_{i=1}^{i=m} S[i]^2 \cdot i^j \bmod p$ for $j \geq 0$.*

▶ **Lemma 7** (Clifford, Kociumaka, and Porat [11]). *Given the sketches $sk_k(S_1)$ and $sk_k(S_2)$ of two strings of equal lengths $\ell \leq m$, in $\tilde{\mathcal{O}}(k)$ time one can decide (with high probability) whether the Hamming distance between S_1 and S_2 is at most k. If so, one can report each mismatch p between S_1 and S_2 as well as $S_1[p]$ and $S_2[p]$. The algorithm uses $\mathcal{O}(k)$ space.*

▶ **Lemma 8** (Clifford, Kociumaka, and Porat [11]). *We can construct one of the sketches $sk_k(S_1)$, $sk_k(S_2)$, or $sk_k(S_1S_2)$ given the other two in $\tilde{\mathcal{O}}(k)$ time using $\mathcal{O}(k)$ space, provided that all the processed strings are over the alphabet $[0, p-1]$ and are of length at most m. Furthermore, we can compute $sk_k(S^m)$, where S^m is a concatenation of m copies of S, in $\tilde{\mathcal{O}}(k)$ time as well under the same assumption.*

3 Algorithm based on the randomised k-errata tree

In this section, we show a streaming algorithm for dictionary matching with k mismatches based on a new randomised implementation of the k-errata tree, a data structure introduced by Cole, Gottlieb, and Lewenstein [12].

▶ **Lemma 9.** *There is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(k \cdot (m + d \log^k d))$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character, where occ is the number of the occurrences. On request, the algorithm can output the mismatches in $\tilde{\mathcal{O}}(k)$ time per occurrence. The algorithm is randomised and its answers are correct w.h.p.*

We start by showing a randomised version of the k-errata tree. The k-errata tree [12] is a data structure that supports dictionary look-up with k mismatches queries: Given a query string Q, find all patterns in the dictionary that are at the Hamming distance at most k from it. The k-errata tree is a collection of compact tries that can answer a dictionary look-up with k mismatches for a string Q of length m in time $\mathcal{O}(m + \log^k d \log\log m + occ)$. The query algorithm consists of $\mathcal{O}(\log^k d)$ calls to a procedure called PrefixSearch. This procedure takes three arguments, a compact trie τ, a node u (or a position on an edge) in τ, and a string S, and must find the longest path in τ that starts at u and is labelled by a prefix of S. In our case, τ is always one of the compact tries of the k-errata tree, and S is always one of the suffixes of Q. Cole, Gottlieb, and Lewenstein [12] showed that one can use the suffix tree on the patterns in the dictionary to answer the PrefixSearch queries deterministically in $\mathcal{O}(\log\log m)$ time after $\mathcal{O}(m)$-time shared preprocessing. Unfortunately, this solution uses too much space and time for our purposes. In the randomised version of the k-errata tree, we implement each of the compact tries as a z-fast trie:

▶ **Fact 10** (z-fast tries, Belazzougui et al. [6])**.** *Consider a string S and suppose that we can compute the Karp—Rabin fingerprint of any prefix of S in t_φ time. A compact trie on a set of r strings of length at most m can be implemented in $\mathcal{O}(r)$ space to support the following queries in $\mathcal{O}(t_\varphi \cdot \log m)$ time: Given S, find the highest node v such that the longest prefix of S present in the trie is a prefix of the label of the root-to-v path. The answers are correct w.h.p.*[2]

This gives an efficient implementation of all PrefixSearch queries if u is the root of a compact trie, but there are more details for the general case. We provide full details, as well as the definition of the k-errata tree, in Appendix A.

▶ **Lemma 11.** *A dictionary of d patterns of maximal length m can be preprocessed into a data structure which we call* randomised k-errata tree *that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and allows retrieving all the patterns that are within Hamming distance k from Q or one of its prefixes in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time, assuming that we know the k-mismatch sketches of all prefixes of Q. The answers are correct w.h.p.*

We are now ready to give the proof of Lemma 9.

Proof of Lemma 9. During the preprocessing step, the algorithm builds the k-errata tree for the reverses of the patterns. During the main step, the algorithm maintains the Karp–Rabin fingerprints and the k-mismatch sketches of the m longest prefixes of the text in a round-robin fashion updating them in $\tilde{\mathcal{O}}(k)$ time when a new character arrives (Lemma 8). If the text

[2] Error probability comes from the collision probability for Karp–Rabin fingerprints.

ends with a k-mismatch occurrence of some pattern P_i, there is a suffix of the text of length $|P_i| \leq m$ such that the Hamming distance between it and some pattern in the dictionary is bounded by k. It means that we can retrieve all occurrences of such patterns by using the randomised k-errata tree for the reverse of the m-length suffix of the text. We can retrieve the fingerprint and the k-mismatch sketch of any substring of this suffix in $\tilde{\mathcal{O}}(k)$ time (Lemma 8), and therefore perform the dictionary look-up query in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time. In total, the algorithm uses $\tilde{\mathcal{O}}(k \cdot (m + d \log^k d))$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. ◀

4 Improving space

The algorithm of Corollary 2 is efficient in terms of space, but not in terms of time. The algorithm of Lemma 9 is efficient in terms of time, but not in terms of space. In this section, we show that it is possible to achieve sublinear dependency on m for space, and in m and d for time:

▶ **Theorem 12.** *There is a streaming algorithm that solves the problem of dictionary matching with k mismatches, for any $k \geq 1$, in $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p. Both false-positive and false-negative errors are allowed.*

Note that the time complexity of the algorithm is amortised. In Appendix 5 we show how to de-amortise the running time to obtain our main result, Theorem 1. The techniques that we use have flavour similar to [9–11, 18], but make a significant step forward to allow *both* mismatches and multiple patterns.

▶ **Definition 13** (k-period, Clifford et al. [10])**.** *The k-period of a string $S = S[1]S[2]\dots S[m]$ is the minimal integer $\pi > 0$ such that the Hamming distance between $S[\pi + 1, m]$ and $S[1, m - \pi]$ is at most $2k$.*

▶ **Observation 14.** *If the k-period of S is larger than d, there can be at most one k-mismatch occurrence of S per d consecutive positions of the text.*

Hereafter we assume $k \log \log d < \log^k d$ (all logs are base two), which is true for any $d \geq 3$ and $k \geq 1$. For $d = 1, 2$ we can use Corollary 2 to achieve the complexities of Theorem 1. Furthermore, we assume that the lengths of the patterns are at least $3d$, for shorter patterns we can use the algorithm of Lemma 9. We partition the dictionary into two smaller dictionaries: the first dictionary $\mathcal{D}_1$ contains the patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is larger than d, and the second dictionary $\mathcal{D}_2$ contains patterns P_i such that the k-period of their suffix τ_i is at most d. In Section 4.2 we show a streaming algorithm that finds all k-mismatch occurrences of the patterns in $\mathcal{D}_1$, and in Section 4.3 a streaming algorithm for $\mathcal{D}_2$. We run the two algorithms in parallel to obtain Theorem 12.

4.1 Reminder: The k-mismatch algorithm of Porat and Porat

We first give an outline of the k-mismatch algorithm of Porat and Porat [26] and explain how it can be applied to the dictionary matching setting.

Porat and Porat showed that the k-mismatch problem for a pattern P can be reduced to exact pattern matching in the following way. Let $Q = \{q_1, q_2, \dots, q_{\log m / \log \log m}\}$ be the set of the first $\log m / \log \log m$ primes larger than $\log m$, and $R = \{r_1, r_2, \dots, r_{k \log m / \log \log m}\}$

be the set of the first $k \log m / \log\log m$ primes larger than $\log m$. A subpattern $(P_i)^{\ell}_{q,r}$ of a pattern P_i is defined by two primes $q \in Q, r \in R$ and an integer $1 \le \ell \le q \cdot r$, namely, $P^{\ell}_{q,r} = P[\ell]P[q \cdot r + \ell]P[2q \cdot r + \ell] \dots$ and so on until the end of P. The prime number theorem implies that $q \in \tilde{\mathcal{O}}(1)$ and $r \in \tilde{\mathcal{O}}(k)$, and therefore for a fixed q, r there are $\tilde{\mathcal{O}}(k)$ subpatterns.

▶ **Lemma 15** (Porat–Porat [26]). *Consider an alignment of the pattern P_i and the text. Given the subset of the subpatterns of P_i that match exactly at this alignment, there is a deterministic $\tilde{\mathcal{O}}(k^2)$-time algorithm that outputs "No" if the Hamming distance between P_i and T is larger than k, and the true value of the Hamming distance otherwise.*

Using this reduction, we show a streaming algorithm that uses $\tilde{\mathcal{O}}(k^3 d)$ space and processes each character of the text in $\tilde{\mathcal{O}}(k^2 \log\log(m + d))$ time. On request, the algorithm can tell in $\tilde{\mathcal{O}}(k^2)$ time if there is a k-mismatch occurrence of a pattern P_i that ends at the current position of text. During the preprocessing step, for each pair of primes $q \in Q, r \in R$, we build a compact trie on the reverses of the subpatterns $(P_i)^{\ell}_{q,r}$. Furthermore, we preprocess each trie (using a depth-first traversal) to be able to tell in $\mathcal{O}(1)$ time if the reverse of the subpattern $(P_i)^{\ell}_{q,r}$ is a prefix of the reverse of the subpattern $(P_{i'})^{\ell'}_{q,r}$.

During the main stage, for each pair of primes $q \in Q, r \in R$ and an integer $1 \le \ell \le q \cdot r$ we define a text substream $T^{\ell}_{q,r} = T[\ell]T[q \cdot r + \ell]T[2q \cdot r + \ell] \dots$ and so on until the end of T. We then run the streaming dictionary matching algorithm of Clifford et al. [9] for the substream $T^{\ell}_{q,r}$ and the dictionary of subpatterns $(P_i)^{\ell'}_{q,r}$, where $i = \{1, 2, \dots, d\}$ and $1 \le \ell' \le q \cdot r$. At each position, the streaming dictionary matching algorithm outputs the id of the longest subpattern that matches at this position. In total for each pair of primes there are $\tilde{\mathcal{O}}(k)$ substreams and $\tilde{\mathcal{O}}(kd)$ subpatterns per substream, and therefore the algorithm uses $\tilde{\mathcal{O}}(k^3 d)$ space and $\tilde{\mathcal{O}}(k \log\log(m + kd))$ time per character, the latter is because each time a new character $T[p]$ arrives, where $p = q \cdot r + \ell$, we must update exactly one substream $T^{\ell}_{q,r}$ (and over all $q \in Q, r \in R$, there are $\tilde{\mathcal{O}}(k \log\log(m + kd))$ substreams to update).

Using the compact tries built at the preprocessing step, we can then check, for any subpattern $(P_i)^{\ell}_{q,r}$, if it matches at this position in $\mathcal{O}(1)$ time and therefore can decide if there is a k-mismatch occurrence of P_i in $\tilde{\mathcal{O}}(k^2)$ by Lemma 15.

4.2 Streaming algorithm for patterns with large periods

In this section, we show a streaming algorithm for the dictionary $\mathcal{D}_1$ that contains patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is at least d.

▶ **Lemma 16.** *If for each pattern in the dictionary $\mathcal{D}_1$ the k-period of its $2d$-length suffix is larger than d, then there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p.*

Note that any k-mismatch occurrence of a pattern P_i ends with a k-mismatch occurrence of τ_i. The first step of our algorithm is to retrieve the occurrences of τ_i. To do so, we run the streaming algorithm of Lemma 9. At each position of the text, the algorithm outputs all indices i such that there is a k-mismatch occurrence of τ_i ending at this position. After having found the occurrences of τ_i, our second step is to check if they can be extended into full occurrences of P_i which we do with the help of the streaming algorithm explained in Section 4.1.

We now analyse the complexity of the algorithm. To find occurrences of the suffixes τ_i, we need $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. The algorithm of Section 4.1 uses $\tilde{\mathcal{O}}(k^3 d)$ space and $\tilde{\mathcal{O}}(k \log\log(m + kd))$ time per character. To test if an

occurrence of τ_i can be extended into an occurrence of P_i, we need $\tilde{\mathcal{O}}(k^2)$ time. Importantly, by Observation 14, there is at most one k-mismatch occurrence of τ_i per d positions of the text. Hence, we will need $\tilde{\mathcal{O}}(k^2 d)$ time to test all k-mismatch occurrences of the suffixes τ_i that end at any d consecutive positions of the text. Lemma 16 follows.

4.3 Streaming algorithm for patterns with small periods

In this section, we show a streaming algorithm for the second dictionary $\mathcal{D}_2$ that contains patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is at most d.

▶ **Lemma 17.** *If for each pattern in the dictionary $\mathcal{D}_2$ the k-period of its $2d$-length suffix is smaller than d, then there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p.*

We define τ'_i, $|\tau'_i| \geq |\tau_i|$, to be the longest suffix of P_i with the k-period at most d. Two cases are possible:

1. The suffix τ'_i equals P_i (in other words, the k-period of P_i is at most d);
2. The suffix τ'_i is a proper suffix of P_i.

We first assume that Case 1 holds for all the patterns in $\mathcal{D}_2$, and then extend the algorithm to Case 2 as well. We start by showing a simple but important property of patterns with small periods.

▶ **Lemma 18.** *Consider a position r of the text. Let $j \cdot d$ be the largest multiple of d that is smaller than r and L be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ with the $2k$-period at most d. Every k-mismatch occurrence of $P_i \in \mathcal{D}_2$ in T that ends at the position r is fully contained in $L\,T[j \cdot d + 1, r]$.*

Proof. Consider an occurrence $T[\ell, r]$ of a pattern $P_i \in \mathcal{D}_2$ that ends at the position r. Since the length of P_i is at most m, $\ell \geq r - m + 1 > j \cdot d - m + 1$. Now, let $\rho \leq d$ be the k-period of P_i. Since the Hamming distance between $T[\ell, r]$ and P_i is at most k, the $2k$-period of $T[\ell, r]$ is at most ρ. Indeed, the Hamming distance between $T[\ell + \rho - 1, r]$ and $T[\ell, r - \rho + 1]$ is at most $2k$ plus the Hamming distance between $P_i[\rho, |P_i|]$ and $P_i[1, |P_i| - \rho + 1]$ which can be upper bounded by $2k$ in its turn. Therefore, the $2k$-period of $T[\ell, j \cdot d]$ is at most ρ and hence it is contained in L. ◀

4.3.1 Algorithm for Case 1

We are now ready to describe the algorithm for the Case 1. During the preprocessing stage, we build the k-errata tree for the reverses of all the patterns in $\mathcal{D}_2$. During the main stage of the algorithm, we maintain the suffix L and an associated data structure D. The data structure D will be used to answer the following queries in $\tilde{\mathcal{O}}(k)$ time: Given a suffix of L defined by its starting and ending positions, return its $4k$-mismatch sketch.

Let us first explain how we maintain L. We initialize L with an empty string and update it each d characters. While reading the next d characters of the text, that is a substring $T[(j-1) \cdot d + 1, j \cdot d]$, we compute the $4k$-mismatch sketches of its d prefixes in $\tilde{\mathcal{O}}(kd)$ time (Lemma 8). After having reached $T[j \cdot d]$, we update L. It suffices to compute the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ such that the Hamming distance between it and its copy shifted by ρ positions, for $\rho = 1, \ldots, d$, is at most $2k$. For a fixed value of ρ, we use binary search and the $4k$-mismatch sketches. Suppose we want to decide whether the Hamming distance between $T[\ell, j \cdot d - \rho + 1]$ and $T[\ell + \rho, j \cdot d]$ is at most $4k$. First note that we must only consider the case when $T[\ell, j \cdot d]$ is fully contained in $L\,T[(j-1) \cdot d + 1, j \cdot d]$.

▶ **Observation 19.** *If $T[\ell, j \cdot d]$ is longer than $L\,T[(j-1) \cdot d + 1, j \cdot d]$, then its $2k$-period is larger than d.*

Proof. If the $2k$-period of $T[\ell, j \cdot d]$ is at most d, the $2k$-period of $T[\ell, (j-1) \cdot d]$ is at most d. If $T[\ell, (j-1) \cdot d]$ is longer than L, we obtain a contradiction. ◀

Since we are only interested in the case when $T[\ell, j \cdot d]$ is fully contained in $L\,T[(j-1) \cdot d + 1, j \cdot d]$, both $T[\ell, j \cdot d - \rho + 1]$ and $T[\ell + \rho, j \cdot d]$ can be represented as a concatenation of a suffix of L and a substring of $T[(j-1) \cdot d + 1, j \cdot d]$. We can retrieve the $4k$-mismatch of any suffix of L in $\tilde{\mathcal{O}}(k)$ time using the data structure D and the $4k$-mismatch sketch of any substring of $T[(j-1) \cdot d + 1, j \cdot d]$ using Lemma 8. Therefore, we can compute the $4k$-mismatch sketches of both strings and hence the Hamming distance between them in $\tilde{\mathcal{O}}(k)$ time using Lemma 7. In total, we need $\tilde{\mathcal{O}}(dk)$ time to update L, or $\tilde{\mathcal{O}}(k)$ amortised time per character.

We now define the data structure D and explain how we update it. Suppose that after the latest update the $2k$-period of L is $\rho \leq d$ and consider a partitioning of L into non-overlapping blocks of length ρ. We say that a block contains a mismatch if, for some i, its i-th character is different from the i-th character of the preceding block. For convenience, we also say that the first block in L is mismatch-containing.

▶ **Observation 20.** *The total number of the blocks containing a mismatch is $\mathcal{O}(k)$.*

Proof. By definition, the Hamming distance between $L[1, |L| - \rho + 1]$ and $L[\rho + 1, |L|]$ is at most $4k$, and it upper bounds the number of the blocks containing a mismatch. ◀

D consists of two parts. First, we store a binary search tree on the set of the starting positions of all blocks containing a mismatch. Secondly, for each block $L[(j-1) \cdot \rho + 1, j \cdot \rho]$ containing a mismatch we store the $4k$-mismatch sketch of each of its suffixes, as well as the sketch of the suffix of L that starts at the position $(j-1) \cdot \rho + 1$. In total, D occupies $\tilde{\mathcal{O}}(k^2 d)$ space.

▶ **Lemma 21.** *We can update D in $\tilde{\mathcal{O}}(k^2)$ amortised time per character. After it has been updated, we can compute the $4k$-mismatch sketch of any suffix of L in $\tilde{\mathcal{O}}(k)$ time.*

Proof. Using the $4k$-mismatch sketches for $L[\rho, |L|]$ and $L[1, |L| - \rho + 1]$, we can find the $\mathcal{O}(k)$ blocks containing a mismatch in $\tilde{\mathcal{O}}(k)$ time. We can then re-build the binary search tree in $\tilde{\mathcal{O}}(k)$ time and compute the sketches for the $\mathcal{O}(k)$ mismatch-containing blocks in $\tilde{\mathcal{O}}(k^2 d)$ time.

Given a starting position ℓ of a suffix of L, we use the binary search tree to determine the streak of blocks without mismatches it belongs to, and retrieve the sketch of the suffix starting just after the streak in $\tilde{\mathcal{O}}(k)$ time. The remaining part consists of a number of repetitions of the block containing the position ℓ prepended with the suffix of the block. We can compute the sketch of the block and of its suffix in $\tilde{\mathcal{O}}(k)$ time, and therefore we can compute the sketch of the remaining part in $\tilde{\mathcal{O}}(k)$ time using Lemma 8. ◀

Let $T[r]$ be the latest arrived character of the text. To retrieve the k-mismatch occurrences that end at the position r, we use the k-errata tree for the reverses of the patterns in $\mathcal{D}_2$ that we build during the preprocessing stage. Let $j \cdot d$ be the largest multiple of d that is at most r and let L be defined as above. By Lemma 18, any k-mismatch occurrence of pattern $P_i \in \mathcal{D}_2$ that ends at r must be equal either to a suffix of $T[j \cdot d + 1, r]$, or to the concatenation of some suffix of L and $T[j \cdot d + 1, r]$. The data structure D allows to compute the $4k$-mismatch sketch (and therefore k-mismatch) of any suffix of L in $\tilde{\mathcal{O}}(k)$ time. We can

also compute the $4k$-mismatch sketch of any of the d latest suffixes of the text in $\tilde{\mathcal{O}}(k)$ time. Therefore, we can retrieve the k-mismatch occurrences of the patterns for a current position in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time using the k-errata tree. In total, the algorithm for Case 1 uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character.

4.3.2 Extension to Case 2 and wrapping up

Consider now Case 2. Note first that the $2k$-period of a string $P_i[|P_i| - |\tau_i'|, |P_i|]$, which is τ_i' extended by one character, must be at least d, and therefore by Observation 14 there can be at most one k-mismatch occurrence of $P_i[|P_i| - |\tau_i'|, |P_i|]$ per d positions of the text. We use the techniques of the algorithm for Case 1 to retrieve the occurrences of $P_i[|P_i| - |\tau_i'|, |P_i|]$, and then use the techniques of the algorithm for patterns with large periods (Lemma 16) to extend the retrieved occurrences.

In more detail, consider a position r of the text. As before, let $j \cdot d$ be the largest multiple of d that is smaller than r and L be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ with the $2k$-period at most d. Let now L' be the suffix L extended by one character to the left, i.e. $L' = T[j \cdot d - |L|, j \cdot d]$. By definition, the $(2k+1)$-period of L' is at most $d-1$. Furthermore, similar to Lemma 18, we can show that any k-mismatch occurrence of $P_i[|P_i| - |\pi_i'|, |P_i|]$ ending at the position r must be fully contained in $L'T[j \cdot d + 1, r]$.

Similarly to the previous section, we can maintain L' and the associated data structure D' using $\tilde{\mathcal{O}}(k^2 d)$ space and $\tilde{\mathcal{O}}(k^2 d)$ time per character. Using D', we can compute the $4k$-mismatch (and therefore k-mismatch) sketch of any suffix of $L'T[j \cdot d + 1, r]$ in $\tilde{\mathcal{O}}(k)$ time and hence we can find the occurrences of $P_i[|P_i| - |\tau_i'|, |P_i|]$ using the k-errata tree in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. We now need to decide which of the found occurrences can be extended into full occurrences of P_i. In order to do this, we run the algorithm of Section 4.1. When we find an occurrence of $P_i[|P_i| - |\tau_i'|, |P_i|]$, we test it in $\tilde{\mathcal{O}}(k^2)$ time.

In total, the algorithm for Case 2 uses $\tilde{\mathcal{O}}(kd \log^k d + k^2 d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. Lemma 17 and Theorem 12 follow.

5 Proof of Theorem 1 – de-amortisation

Recall that the streaming algorithm of Theorem 12 is comprised of the algorithms of Lemma 16 and of Lemma 17 ran in parallel. Below we explain how to de-amortise these two algorithms. We use a standard approach called the *tail trick* that was already used in [9–11].

5.1 De-amortised algorithm with a delay

First, note that there is an easy way to de-amortise the algorithm of Lemma 16 if we allow delaying the occurrences by d characters. In order to do that, we divide the text into non-overlapping blocks of length d, and de-amortise the processing time of a block over the next block, by running $\tilde{\Theta}(k + \log d)$ steps of the computation per character. We will need to memorize the occurrences that end at the last $2d$ positions of the text, but this requires only $\mathcal{O}(d)$ space and we can afford it.

We now show how to de-amortise the algorithm for Case 1 of Lemma 17. This time, we will not need the delay. The only step of the algorithm that requires de-amortisation is updating L and D. We can de-amortise this step in a standard way. Namely, we de-amortise the time we need for an update by running $\tilde{\Theta}(k \log d)$ steps of the computation per each of the next d characters of text. We also maintain the sketches of the $2d$ longest prefixes of the text in a round-robin fashion using $\tilde{\mathcal{O}}(kd)$ space and $\mathcal{O}(k)$ time. If we need to extract the

sketch of some suffix of L before the update is finished, we use the previous version of the data structure and the sketches of the $2d$ latest suffixes of the text to compute the required values using Lemma 8.

Finally, we show how to de-amortise the algorithm of Case 2 of Lemma 17, again with a delay of d characters. Recall that this algorithm first finds the k-mismatch occurrences of the suffixes $P_i[|P_i| - |\tau_i'|, |P_i|]$ using an algorithm similar to the algorithm for Case 1 of Lemma 17, which can be de-amortised with no delay as explained above, and then tests these occurrences using the algorithm of Section 4.1, which can be de-amortised with a delay of d characters. Importantly, there are at most d occurrences that need to be tested per d characters, so we can memorize them until we can test them. The claim follows.

5.2 Removing the delay

We now show how to remove the delay. Recall that we assume the patterns to have lengths larger than $3d$. We partition each pattern $P_i = H_i\, Q_i$, where Q_i is the suffix of P_i of length d, and H_i is the remaining prefix. The idea is to find occurrences of the prefixes H_i and of the suffixes Q_i independently, and then to see which of them form an occurrence of P_i.

As above, we have three possible cases: the k-period of $H_i[|H_i| - 2d + 1, |H_i|]$ is larger than d; the k-period of H_i is at most d; the k-period of H_i is larger than d but the k-period of $H_i[|H_i| - 2d + 1, |H_i|]$ is at most d.

In the second case, we do not need to change much. For the current position r of the text we consider the largest $j \cdot d$ such that $r - j \cdot d \geq d$ and define L to be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ such that its $2k$-period is at most d. We store the k-errata tree on the reverses of $P_i = H_i\, Q_i$ and run the de-amortised algorithm described in the previous section that maintains the suffix L. Any k-mismatch occurrence of a pattern P_i is fully contained in the concatenation of L and a suffix of the text of length $3d$, and therefore we can find all such occurrences using the k-errata tree as above.

We now explain how we remove the delay in the first and third cases. To find the occurrences of Q_i we use the streaming algorithm of Lemma 9. To find the occurrences of H_i we use the de-amortised version of the algorithm of Lemma 16 or of Lemma 17, as appropriately, that report the occurrences with a delay of at most d characters. It means that at the time when we find an occurrence of Q_i, the corresponding occurrence of H_i is already reported, so it is easy to check whether they form an occurrence of P_i. The only technicality is that we need to store the occurrences of H_i that we found while processing the last d characters of the text.

To this end, we use a dynamic hashing scheme [15]. The scheme allows to store a dynamic dictionary in linear space and with high probability guarantees constant look-up and update times. The answers to the look-up queries are always correct. Note that we can modify the data structure slightly to have constant time per operation if we allow the answers to be correct only with high probability (which we can afford), namely, if an operation takes too much time, we can simply abandon it.

We use the scheme for each of the last d positions of the text. Namely, consider a position p of the text and suppose that we found a set of k-mismatch occurrences of the prefixes H_i that end at p. Consider one of the prefixes, H_i, and let the Hamming distance between a prefix H_i and the text be $h \leq k$. Recall that by Fact 23 there are $\mathcal{O}(\log^k d)$ nodes of the k-errata tree labelled by Q_i. For each such node u of the k-errata tree, we insert a pair (u, h) into the dictionary. In case we insert a pair (u, h) several times for different prefixes H_i's, we associate (u, h) with the set of such prefixes. Note that at any moment the total size of the dictionaries is $\tilde{\mathcal{O}}(d \log^k d)$ as each of the patterns H_i has at most one k-mismatch occurrence over each d consecutive positions of the text.

Suppose we are at a position p of the text and we have run a dictionary look-up query and found the $\mathcal{O}(\log^k d)$ nodes in the tries of the k-errata tree corresponding to the suffixes Q_i that occur at this position with at most k mismatches. For each such node u we know the Hamming distance h' between the occurrences and the text. We then go to the dictionary at the position $(p - 2d)$ and look up pairs $(u, k - h'), (u, k - h' - 1), \ldots, (u, 0)$. If they are in the dictionary, we report all H_i's associated with these pairs. This step takes $\mathcal{O}(k \log^k d + occ)$ time.

6 Proof of Lemma 3 – space lower bound

In the communication complexity setting the Index problem is stated as follows. We assume that there are two players, Alice and Bob. Alice holds a binary string of length n, and Bob holds an index i encoded in binary. In a one-round protocol, Alice sends Bob a single message (depending on her input and on her random coin flips) and Bob must compute the i-th bit of Alice's input using her message and his random coin flips correctly with probability $> 2/3$. The length of Alice's message (in bits) is called the randomised one-way communication complexity of the problem. The randomised one-way communication complexity of the Index problem is $\Omega(n)$ [23].

Given a streaming algorithm for dictionary matching with k mismatches, we can construct a randomised one-way communication complexity protocol for the Index problem as follows. As above, let d be the size of the dictionary, and assume that $n = kd$. Split Alice's string into d blocks of length k. Let $\#, \$, \$_1, \ldots, \$_d$ be distinct characters different from $\{0, 1\}$. For the j-th block B_j create a string $P_j = (\$_j)^{k+1}\#B_j$, where $(\$_j)^{k+1}$ means that we repeat the character $\$_j$ $(k + 1)$ times. For Bob's input $i = k \cdot q + r$ we create a string T which is equal to $(\$_q)^{k+1}$ concatenated with a string of length $k + 1$ obtained from $\$^{k+1}$ by changing the $(r + 1)$-th bit to 0. A streaming dictionary matching with k mismatches for the set of patterns P_i and T will output a k-mismatch occurrence of B_q at the position $2k + 2$ of the text iff the r-th bit of Alice's input is equal to 0. Therefore, if Alice preprocesses P_j as in the streaming algorithm and sends the result to Bob, Bob will be able to continue to run the streaming algorithm on T to decide the i-th bit of Alice's input. Therefore, the lower bound for communication complexity of the Index problem is a space lower bound for any streaming algorithm for dictionary matching with mismatches. Lemma 3 follows.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communincations of the ACM*, 18(6):333–340, June 1975. doi:10.1145/360825.360855.

2 Ricardo Baeza-Yates and Gonzalo Navarro. Multiple approximate string matching. In *Proc. of the 5th Workshop on Algorithms and Data Structures*, pages 174–184, 1997. doi:10.1007/3-540-63307-3_57.

3 Djamal Belazzougui. Succinct Dictionary Matching with No Slowdown. In *Proc. of the 21st Annual Symposium on Combinatorial Pattern Matching*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

4 Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *Journal of Discrete Algorithms*, 14:91–106, 2012. doi:10.1007/978-3-642-19222-7_10.

5 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone Minimal Perfect Hashing: Searching a Sorted Table with $O(1)$ Accesses. In *Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 785–794, 2009. doi:10.1137/1.9781611973068.86.

6 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic Z-Fast Tries. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval*, pages 159–172, 2010. doi:10.1007/978-3-642-16321-0_15.

7 Djamal Belazzougui and Mathieu Raffinot. Average Optimal String Matching in Packed Strings. In *Proc. of the 8th International Conference on Algorithms and Complexity*, pages 37–48, 2013. doi:10.1007/978-3-642-38233-8_4.

8 Dany Breslauer and Zvi Galil. Real-Time Streaming String-Matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, August 2014. doi:10.1145/2635814.

9 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary Matching in a Stream. In *Proc. of the 23rd Annual European Symposium on Algorithms*, pages 361–372, 2015. doi:10.1007/978-3-662-48350-3_31.

10 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In *Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.

11 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proc. of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1125, 2019. doi:10.1137/1.9781611975482.68.

12 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. of the 36th Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004. doi:10.1145/1007352.1007374.

13 Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of the 6th International Colloquium on Automata, Languages and Programming*, pages 118–132, 1979. doi:10.1007/3-540-09510-1_10.

14 Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3):107–113, 1999. doi:10.1016/S0020-0190(99)00092-7.

15 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. *Dynamic Hashing in Real Time*, pages 95–119. Vieweg+Teubner Verlag, Wiesbaden, 1992. doi:10.1007/978-3-322-95233-2_7.

16 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via Small-Space Multiple-Pattern Matching. In *Proc. of the 23rd European Symposium on Algorithms*, pages 533–544, 2015. doi:10.1007/978-3-662-48350-3_45.

17 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards Optimal Approximate Streaming Pattern Matching by Matching Multiple Patterns in Multiple Streams. In *Proc. of the 45th International Colloquium on Automata, Languages, and Programming*, pages 65:1–65:16, 2018. doi:10.4230/LIPIcs.ICALP.2018.65.

18 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In *Proc. of the 25th Annual European Symposium on Algorithms*, volume 87, pages 41:1–41:15, 2017. doi:10.4230/LIPIcs.ESA.2017.41.

19 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster Compressed Dictionary Matching. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval*, pages 191–200, 2010. doi:10.1007/978-3-642-16321-0_19.

20 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987. doi:10.1147/rd.312.0249.

21 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In *Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching*, volume 54, pages 6:1–6:13, 2016. doi:10.4230/LIPIcs.CPM.2016.6.

22 Dmitry Kosolobov and Nikita Sivukhin. Compressed Multiple Pattern Matching. *CoRR*, abs/1811.01248, 2018. arXiv:1811.01248.

23 Ilan Kremer, Noam Nisan, and Dana Ron. On Randomized One-round Communication Complexity. In *Proc. of the 27th Annual ACM Symposium on Theory of Computing*, pages 596–605, 1995. doi:10.1007/s000370050018.
24 Robert Muth and Udi Manber. Approximate multiple string search. In *Proc. of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 75–86, 1996. doi:10.1007/3-540-61258-0_7.
25 Gonzalo Navarro. Multiple Approximate String Matching by Counting. In *Proc. of the 4th South American Workshop on String Processing*, pages 95–111, 1997.
26 Benny Porat and Ely Porat. Exact And Approximate Pattern Matching In The Streaming Model. In *Proc. of the 50th Annual Symposium on Foundations of Computer Science*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.
27 Sun Wu and Udi Manber. Agrep - A Fast Approximate Pattern-Matching Tool. In *Proc. of the USENIX Technical Conference*, pages 153–162, 1992.

A Proof of Lemma 11 – randomised k-errata tree

We will first remind the definition of the k-errata tree of Cole et al. [12], and then show a randomised implementation of this data structure.

A.1 Reminder: the k-errata tree

Consider a dictionary $\mathcal{D}$ of d patterns of maximal length m. We start with the compact trie $\mathcal{T}$ for the dictionary $\mathcal{D}$, and decompose it into heavy paths.

▶ **Definition 22.** *The heavy path of $\mathcal{T}$ is the path that starts at the root of $\mathcal{T}$ and at each node v on the path branches to the child with the largest number of leaves in its subtree (*heavy *child), with ties broken arbitrarily. The heavy path decomposition is defined recursively, namely, it is defined to be a union of the heavy path of $\mathcal{T}$ and the heavy path decompositions of the off-path subtrees of the heavy path.*

During the recursive step, we construct a number of new compact tries. For each heavy path H, and for each node $u \in H$ consider the off-path trees hanging from u. First, we create a *vertical substitution trie* for u. Let a be the first character on the edge $(u, v) \in H$. Consider an off-path tree hanging from u, and let $b \neq a$ be the first character on the edge from u to this tree. For each pattern in this off-path tree, we replace b by a. We consider a set of patterns obtained by such a substitution for all off-path trees hanging from u and build a new compact trie for this set. Next, we create *horizontal substitution tries* for the node u. We create a separate horizontal substitution trie for each off-path tree hanging from u. To do so, we take the patterns in it and cut off the first characters up to and including the first character on the edge from u to this tree, and then build a compact trie on the resulting set of patterns. To finish the recursive step we build the $(k-1)$-errata trees for each of the new vertical and horizontal tries.

From the construction, it follows that the k-errata tree is a set of compact tries, and each string S in the tries originates from a pattern in the dictionary $\mathcal{D}$. We mark the end of the path labelled by S by the id of the pattern it originates from.

Queries. A dictionary look-up with k mismatches for a string Q is performed in a recursive way as well. We will make use of a procedure called PrefixSearch. This procedure takes three arguments: a compact trie, a starting node u (or a position on an edge) in this trie, and a query string Q', and must output a pointer to the end of the longest path starting at u and labelled by a prefix of Q'. For the purposes of recursion, we introduce a mismatch credit –

the number of mismatches that we are still allowed to make. We start with the mismatch credit $\mu = k$. The algorithm first runs a PrefixSearch in the trie $\mathcal{T}$ for the query string Q starting from the root. If $\mu = 0$ and the path is labelled by Q, the algorithm returns the ids of the patterns in $\mathcal{D}$ that are associated with the end of the path. Otherwise, we consider the heavy paths $H_1, H_2, \ldots, H_j$ traversed by the PrefixSearch. Let u_i be the position where the PrefixSearch leaves the heavy path H_i, $1 \leq i \leq j$. Note that for $i < j$, u_i is necessarily a node of $\mathcal{T}$, and for $i = j$ it can be a position on an edge. We can divide all the patterns in $\mathcal{D}$ into four groups: (I) Patterns hanging off some node u in a heavy path H_i, where u is located above u_i, $1 \leq i \leq j$; (II) Patterns in the subtrees of u_i's children not in the heavy path H_{i+1}, for $1 \leq i < j$; (III) Patterns in the subtree of the position in H_j that is just below u_j; (IV) If u_j is a node, then patterns in the subtrees of u_j's children not in the heavy path H_j.

We process each of these groups of patterns independently. Consider a pattern P in group I, and let it hang from a node $u \in H_i$, where u is above u_i. Let ℓ be the length of the label of u, then Q and any pattern P in this subtree have a mismatch at the position $\ell + 1$. When creating vertical substitution tries, we removed this mismatch. Therefore, we can retrieve all such patterns that are at the Hamming distance $\leq k$ from Q by running the algorithm recursively with mismatch credit $\mu - 1$ in the $(k-1)$-errata tree that we created for the vertical substitution trie for the node u. The patterns of groups II and IV are processed in a similar way but using the $(k-1)$-errata trees for the horizontal substitution trees. Finally, to process the patterns of group III, we run the algorithm with mismatch credit $\mu - 1$ starting from the position that follows u_j in H_j.

This algorithm correctly retrieves the subset of the patterns in $\mathcal{D}$ that are at Hamming distance $\leq k$ from Q but can be slow as it makes many recursive calls. Cole et al. showed that the number of recursive calls can be reduced to logarithmic by introducing grouping on the substitution tries. In more detail, for each heavy path we consider its vertical substitution tries and build a weight-balanced tree, where the leaves of the weight-balanced tree are the vertical substitution tries, in the top-down order, and for each node of the tree, we create a new trie by merging the tries below it. For each of these group vertical substitution tries we build the $(k-1)$-errata tree. We group the horizontal substitution tries in a similar way, namely, we consider each node u and build a weight-balanced tree on the horizontal substitution tries that we created for the node u.

▶ **Fact 23** (Cole et al. [12]). *The id of any pattern in $\mathcal{D}$ occurs in the compact tries of the k-errata tree $\mathcal{O}(\log^k d)$ times, and as a corollary the total size of the tries is $\mathcal{O}(d \log^k d)$.*

To speed up the algorithm, we search a logarithmic number ($\mathcal{O}(\log d)$) of group substitution tries instead of searching each substitution trie individually. In total, we run $\mathcal{O}(\log^k d)$ PrefixSearch operations.

▶ Remark 24. We will use the k-errata tree to retrieve the patterns that are within Hamming distance k from the query string Q or from one of its prefixes. Recall that we mark each node of the k-errata tree corresponding to an end of a dictionary pattern. Furthermore, during the preprocessing step, we compute a pointer from each node to its nearest marked ancestor. At the end of each PrefixSearch we follow the pointers and retrieve the patterns corresponding to the marked nodes between the end and the start of the PrefixSearch. The number of the PrefixSearch operations that we perform does not change.

It remains to explain how we perform the PrefixSearch operations. Cole et al. gave a deterministic implementation of PrefixSearch that requires $\mathcal{O}(md)$ extra space and $\mathcal{O}(m)$ time of preprocessing, which is too much for our purposes. In the next section, we will show a randomised implementation of PrefixSearch which requires both less space and less time.

A.2 Randomised implementation of the k-errata tree

Recall from above that the k-errata tree is a collection of compact tries. In the randomised version of the k-errata tree, we replace each of them with a z-fast trie (see Fact 10). We also store the k-mismatch sketch of the label of every node of the tries, which requires $\tilde{\mathcal{O}}(kd \log^k d)$ space in total.

We now explain how we answer dictionary look-up with k mismatches. Recall that each dictionary look-up with k mismatches is a sequence of calls to the PrefixSearch procedure, and therefore it suffices to give an efficient implementation of PrefixSearch. We first explain how to implement this operation if it starts at the root of some compact trie of the k-errata tree. Assuming that we can retrieve the Karp–Rabin fingerprint of any substring of Q in $\mathcal{O}(1)$ time, Fact 10 immediately implies that a PrefixSearch starting at the root of a compact trie can be implemented in $\mathcal{O}(\log m)$ time. Note that if the end of the PrefixSearch is a position on an edge of the trie, then the functionality of the z-fast tries will allow us retrieving only the edge this position belongs to, but not the position itself. As we show below, it is sufficient for our purposes.

We now give an implementation of a PrefixSearch starting at an arbitrary position of a compact trie by reducing it first to a PrefixSearch that starts at a node of the trie and then to a PrefixSearch that starts at the root of the trie. We first show a reduction from a PrefixSearch that starts at an arbitrary position on an edge to a PrefixSearch that starts at a node. As we explained above, we might know the edge this starting position belongs to, but the position itself. However, from the description of the query algorithm in Section A.1 it follows that the algorithm will continue along the edge by running PrefixSearch operations until it either runs out of the mismatch credit or reaches the lower end of the edge. We will fast-forward to the lower end of the edge using the k-mismatch sketches. Namely, let Q' be the query string when we entered the current tree (note that we do not change the tree when retrieving patterns of group III). Importantly, the string Q' is a suffix of Q. We want to check whether we can reach the lower end of the edge and not run out of the mismatch credit. In other words, we want to compare the number of mismatches between the label S of the lower end of the edge and the prefix S' of Q' of length $|S|$, and the mismatch credit. We use the k-mismatch sketches for this task. We store the sketch of S, and the sketch of S' can be computed in $\tilde{\mathcal{O}}(k)$ time as it is a substring of Q. Having computed the sketches, we can compute the Hamming distance between S and S' using Lemma 7. If the Hamming distance is larger than the available mismatch credit, we stop, otherwise, we continue the PrefixSearch from the lower end of the edge.

Finally, we show an implementation of a PrefixSearch for a string Q' that starts at a node u of a trie. Let S be the label of u. Our task is equivalent to performing a PrefixSearch starting from the root of a trie for a string $S\,Q'$. Recall that Fact 10 assumes that we can extract the Karp–Rabin fingerprint of any prefix of $S\,Q'$. We do not know the Karp–Rabin fingerprints of the prefixes of $S\,Q'$, but we can compute them as follows. First, we use the k-mismatch sketches similar to above to compute the at most k mismatches that occurred on the way from the root of the trie to u. After having computed the mismatches, we can compute any of the fingerprints in $\tilde{\mathcal{O}}(k)$ time by taking the fingerprint of the corresponding substring of Q and "fixing" it in at most k positions.

So, we can answer a dictionary look-up with k mismatches query in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time, and to compute the mismatches for each of the retrieved patterns in $\tilde{\mathcal{O}}(k)$ time per pattern if requested.