Approximating Text-to-Pattern Hamming Distances*

Timothy M. Chan
University of Illinois
Urbana-Champaign, Illinois, USA
tmc@illinois.edu

Shay Golan
Bar-Ilan University
Ramat Gan, Israel
golansh1@cs.biu.ac.il

Tomasz Kociumaka
Bar-Ilan University
Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Tsvi Kopelowitz
Bar-Ilan University
Ramat Gan, Israel
kopelot@gmail.com

Ely Porat
Bar-Ilan University
Ramat Gan, Israel
porately@cs.biu.ac.il

ABSTRACT

We revisit a fundamental problem in string matching: given a pattern of length m and a text of length n, both over an alphabet of size σ, compute the Hamming distance (i.e., the number of mismatches) between the pattern and the text at every location. Several randomized $(1+\varepsilon)$-approximation algorithms have been proposed in the literature (e.g., by Karloff (Inf. Proc. Lett., 1993), Indyk (FOCS 1998), and Kopelowitz and Porat (SOSA 2018)), with running time of the form $O(\varepsilon^{-O(1)} n \log n \log m)$, all using fast Fourier transform (FFT). We describe a simple randomized $(1+\varepsilon)$-approximation algorithm that is faster and does not need FFT. Combining our approach with additional ideas leads to numerous new results (all Monte-Carlo randomized) in different settings:

(1) We design the first truly *linear-time* approximation algorithm for constant ε; the running time is $O(\varepsilon^{-2} n)$. In fact, the time bound can be made slightly sublinear in n if the alphabet size σ is small (by using bit packing tricks).

(2) We apply our approximation algorithms to design a faster *exact* algorithm computing all Hamming distances up to a threshold k; its runtime of $O\left(n + \min\left(\frac{nk\sqrt{\log m}}{\sqrt{m}}, \frac{nk^2}{m}\right)\right)$ improves upon previous results by logarithmic factors and is linear for $k \le \sqrt{m}$.

(3) We alternatively design approximation algorithms with better *ε-dependence*, by using fast rectangular matrix multiplication. In fact, the time bound is $O(n \operatorname{polylog} n)$ when the pattern is sufficiently long, i.e., $m \ge \varepsilon^{-c}$ for a specific constant c. Previous algorithms with the best ε-dependence require $O(\varepsilon^{-1} n \operatorname{polylog} n)$ time.

(4) When k is not too small, we design a truly *sublinear-time* algorithm to find all locations with Hamming distance approximately (up to a constant factor) less than k, in time $O((n/k^{\Omega(1)} + occ) n^{o(1)})$, where *occ* is the output size. The algorithm leads to a *property tester* for pattern matching that costs $O((\delta^{-1/3} n^{2/3} + \delta^{-1} \frac{n}{m}) \operatorname{polylog} n)$ time and, with high probability, returns true if an exact match exists and false if the Hamming distance is more than δm at every location.

(5) We design a *streaming* algorithm that approximately computes the Hamming distance for all locations with the distance approximately less than k, using $O(\varepsilon^{-2}\sqrt{k} \operatorname{polylog} n)$ space. Previously, streaming algorithms were known for the exact problem with $O(k \operatorname{polylog} n)$ space (which is tight up to the $\operatorname{polylog} n$ factor) or for the approximate problem with $O(\varepsilon^{-O(1)}\sqrt{m} \operatorname{polylog} n)$ space. For the special case of $k = m$, we improve the space usage to $O(\varepsilon^{-1.5}\sqrt{m} \operatorname{polylog} n)$.

CCS CONCEPTS

• **Theory of computation → Pattern matching**; **Sketching and sampling**.

KEYWORDS

Hamming distance, pattern matching, sampling, sublinear, streaming, property testing

ACM Reference Format:
Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. 2020. Approximating Text-to-Pattern Hamming Distances. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357713.3384266

1 INTRODUCTION

We study a fundamental problem in string matching: given a pattern of length m and a text of length n over an alphabet of size σ, compute the Hamming distance (i.e., the number of mismatches) between the pattern and the text at every location. Of particular interest is the version with a fixed threshold, known as the *k-mismatch* problem: compute the Hamming distances only for locations with distances at most a given value k. The k-mismatch problem can be relaxed to testing for each text location whether the Hamming distance is at most k and, even further, to deciding whether there exists a location with at most k mismatches.

The problem has an extensive history spanning over four decades; see Table 1 for a summary. For arbitrary σ, the best time bound,

*This work was supported in part by ISF grants no. 1278/16 and 1926/19, by a BSF grant 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC '20, June 22–26, 2020, Chicago, IL, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384266

Table 1: Time bounds of known exact algorithms for computing all distances at most k.

Fischer and Paterson [22]	$O(\sigma n \log m)$	Cole and Hariharan [19]	$O\left(n + \frac{nk^4}{m}\right)$
Abrahamson [1]	$O(n\sqrt{m \log m})$	Amir, Lewenstein, and Porat [4]	$O(n\sqrt{k \log k})$
Landau and Vishkin [32, 34]	$O(nk)$	Amir, Lewenstein, and Porat [4]	$O\left(n \log k + \frac{nk^3 \log k}{m}\right)$
Galil and Giancarlo [23]	$O(nk)$	Clifford, Fontaine, Porat, Sach, and Starikovskaya [16]	$O\left(n \operatorname{polylog} m + \frac{nk^2 \log k}{m}\right)$
Sahinalp and Vishkin [38]	$O\left(n + \frac{nk^{O(1)}}{m}\right)$	Gawrychowski and Uznański [24]	$O\left(n \log^2 m \log \sigma + \frac{nk\sqrt{\log n}}{\sqrt{m}}\right)$

$\tilde{O}\left(n + \frac{nk}{\sqrt{m}}\right)$,[1] by Gawrychowski and Uznański (ICALP 2018) [24], subsumes all previous bounds up to logarithmic factors; their paper also provides conditional lower bounds suggesting that no substantially faster "combinatorial" algorithms are possible.

As a function of n, the time bound is $\tilde{O}(n^{3/2})$ in the worst case, when m and k are linear in n. To design faster algorithms, researchers have turned to the *approximate* version of the problem: finding values that are within a $1 + \varepsilon$ factor of the true distances.

Several efficient randomized (Monte-Carlo) algorithms for approximating all Hamming distances have been proposed. There are three main simple approaches:

- Karloff [28] designed an $O(\varepsilon^{-2} n \log n \log m)$-time algorithm, by randomly mapping the alphabet to $\{0, 1\}$, thereby reducing the problem to $O(\varepsilon^{-2} \log n)$ instances with $\sigma = 2$. Each such instance can be solved in $O(n \log m)$ time by standard convolution, i.e., fast Fourier transform (FFT). Karloff's approach can be derandomized (in $O(\varepsilon^{-2} n \log^3 m)$ time, via ε-biased sample spaces or error-correcting codes).
- Indyk [26] solved the approximate decision problem for a fixed threshold in $O(\varepsilon^{-3} n \log n)$ time, by using random sampling and performing $O(\varepsilon^{-3} \log n)$ convolutions in $\mathbb{F}_2$, each doable in $O(n)$ time by a bit-packed version of FFT. The general problem can then be solved by examining logarithmically many thresholds, in $O(\varepsilon^{-3} n \log n \log m)$ time.
- Kopelowitz and Porat [31] designed an $O(\varepsilon^{-1} n \log n \log m)$-time algorithm, by randomly mapping the alphabet to an alphabet of size $O(\varepsilon^{-1})$, thereby reducing the problem to $O(\log n)$ instances with $\sigma = O(\varepsilon^{-1})$. Each such instance can be solved by $O(\varepsilon^{-1})$ convolutions. This result is notable for its better ε-dependence; earlier, Kopelowitz and Porat [30] gave a more complicated algorithm (also based on FFT) with $O(\varepsilon^{-1} n \log n \log m \log \sigma \log(1/\varepsilon))$ randomized running time.

All three algorithms require $O(n \log^2 n)$ time as a function of n, and they all use FFT. Two natural questions arise: (i) can the $n \log^2 n$ barrier be broken? (ii) is FFT necessary for designing nearly linear time algorithms?

1.1 A New Simple Approximation Algorithm

In Sections 3 and 4, we present a randomized approximation algorithm which costs $O(\varepsilon^{-2.5} n \log^{1.5} n)$ time and does not use FFT, thereby answering both questions. As in previous randomized algorithms, the algorithm is Monte-Carlo and its results are correct with high probability, i.e., the error probability is $O(n^{-c})$ for an arbitrarily large constant c.

[1] Throughout the paper, $\tilde{O}$ hides polylogarithmic factors, and $\hat{O}$ hides $n^{o(1)}$ factors. Additionally, $\tilde{O}_\varepsilon$ and $\hat{O}_\varepsilon$ may hide $\varepsilon^{-O(1)}$ factors.

Our approach is based on random sampling (like Indyk's [26]): the Hamming distance is estimated by checking mismatches at a random subset of positions. In order to avoid FFT, our algorithm uses a random subset with more structure: the algorithm picks a random prime p (of an appropriately chosen size) and a random offset b, and considers a subset of positions $\{b, b + p, b + 2p, \ldots\}$. The structured nature of the subset enables more efficient computation. It turns out that even better efficiency is achieved by using multiple (but still relatively few) offsets. When approximating the Hamming distance of the pattern at subsequent text locations, the set of sampled positions in the text changes, and so a straightforward implementation seems too costly. To overcome this challenge, a key idea is to shift the sample a few times in the pattern and a few times in the text (namely, for a tradeoff parameter z, our algorithm considers z shifts in the pattern and p/z shifts in the text).

While these simple ideas individually may have appeared before in one form or another in the literature, we demonstrate that they are quite powerful when put together in the right way, and with a careful choice of parameters—numerous new consequences follow, as we outline below.

1.2 Consequences

A linear-time approximation algorithm. By combining the basic new algorithm with existing (more complicated) techniques, we show that the $O(\varepsilon^{-2.5} n \log^{1.5} n)$ time bound can be further reduced all the way down to *linear* (in n)! More precisely, the new (randomized) time bound is $O(\varepsilon^{-2} n)$. Linear-time algorithms were not known before, even for the approximate decision problem with a fixed threshold, and even in the binary case ($\sigma = 2$). In fact, our final time bound is $O(\frac{n \log \sigma}{\log n} + \frac{n \log^2 \log n}{\varepsilon^2 \log n})$, which is *slightly sublinear* in n when σ is small ($\sigma = n^{o(1)}$).

As the reader may surmise, bit-packing techniques are needed (we assume that the input strings are given in $O(\frac{n \log \sigma}{\log n})$ words). To ease the description, in Section 8, we present a version with $O(\varepsilon^{-2} n \log \log n)$ running time and no messier bit-packing tricks. The final algorithm is provided in the full version of the paper [14].

An improved exact algorithm. We apply our linear-time approximation algorithm to design a faster algorithm for the *exact* k-mismatch problem (computing exactly all distances not exceeding k). The new time bound is $O\left(n + \min\left(\frac{nk\sqrt{\log m}}{\sqrt{m}}, \frac{nk^2}{m}\right)\right)$, which shaves off some logarithmic factors from Gawrychowski and Uznański's result [24] (although, to be fair, their result is deterministic). In particular, the running time is linear when $k \le \sqrt{m}$. Our description (see Section 7) does not rely on Gawrychowski and Uznański's

and is arguably simpler, using *forward differences* [17] to handle approximately periodic patterns.

Improved ε-dependence. Apart from shaving off $\log n$ factors, our approach, combined with fast rectangular matrix multiplication (interestingly), leads to approximation algorithms with improved $\varepsilon^{-O(1)}$ factors in the time cost. As mentioned, Kopelowitz and Porat [30, 31] previously designed algorithms with a factor of ε^{-1}, which improve upon earlier methods with an ε^{-2} factor. We are able to obtain even better ε-dependence in many cases (see Section 9). The precise time bound as a function of m, n, and ε is tedious to state (as it relies on the time complexity of fast rectangular matrix multiplication). Remarkably, when the pattern is sufficiently long, for example, when $m \geq \varepsilon^{-28}$ (the exponent 28 has not been optimized), the running time is actually $O(n \operatorname{polylog} n)$ *without any* $\varepsilon^{-O(1)}$ *factors.*

Sublinear-time algorithms. We also show that truly *sublinear-time* (randomized) algorithms are possible for the approximate decision problem (finding all locations with distances approximately less than k) when the threshold k is not too small, the approximation factor is a constant, and the number *occ* of occurrences to report is sublinear. Such sublinear-time algorithms are attractive from the perspective of big data, as not all of the input needs to be read. All we assume is that the input strings are stored in arrays supporting random access. For example, for an approximation factor $1+\varepsilon$, we obtain a time bound of $\tilde{O}\big(n/k^{\Omega(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))} + occ \cdot k^{O(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))}\big)$, and for an approximation factor near 2, we obtain a time bound of $\hat{O}(n^{4/5} + n/k^{1/4} + occ)$. Different tradeoffs are possible, as the time bound relies on known results on approximate nearest neighbor search in high dimensions (see Section 10). The term *occ* disappears if we just want to decide existence or report one location.

In particular, we obtain a *property tester* for pattern matching: with good probability, the test returns true if an exact match exists, and false if the pattern is δ*-far* from occurring in the text, i.e., its Hamming distance is more than δm at every location. The running time is $\tilde{O}(\delta^{-1/3} n^{2/3} + \delta^{-1} \frac{n}{m})$ (approximate nearest neighbor search is not needed here, and the algorithm is simple). We are not aware of such a property tester for pattern matching, despite the extensive literature on property testing and sublinear-time algorithms, and on the classical pattern matching problem. (In fact, property testing for approximate pattern matching was recently mentioned as an open problem in a paper by Ben-Eliezer et al. [10].)

We remark that some previous work has focused on sublinear-time algorithms with the added assumption that the input strings are generated from a known distribution [5, 15]. By contrast, our results hold for *worst-case* inputs. Additional work considers sublinear-time algorithms for edit-distance problems [7–9]. Some of these sublinear-time algorithms (particularly, by Andoni et al. [5] and Batu et al. [9]) share rough similarities with our general approach.

Streaming approximation algorithms. Another setting where our approach leads to new results is that of (one-pass) *streaming* algorithms. Characters from the text arrive in a stream one at a time, and locations with Hamming distance at most k need to be identified as soon as their last characters are read. Algorithms should use limited (sublinear) space and low processing time per character.

A seminal paper by Porat and Porat [37] provided a streaming algorithm for exact pattern matching ($k = 0$) working in $\tilde{O}(1)$ space and taking $\tilde{O}(1)$ time per text character (Breslauer and Galil [11] subsequently improved the time cost to $O(1)$). Porat and Porat [37] also introduced the first streaming algorithm for the exact k-mismatch problem using $\tilde{O}(k^2)$ time per character and $\tilde{O}(k^3)$ space. Subsequent improvements [16, 25] culminated in an algorithm by Clifford et al. [17], which solves the streaming exact k-mismatch problem in $\tilde{O}(\sqrt{k})$ time per character using $\tilde{O}(k)$ space (this space consumption is optimal regardless of the running time).

Streaming algorithms for the approximate k-mismatch problem have also been considered [16]. However, the only known result not subsumed by the above-mentioned exact algorithm is by Clifford and Starikovskaya [18], who gave a streaming algorithm with $\tilde{O}(\varepsilon^{-5}\sqrt{m})$ space and $\tilde{O}(\varepsilon^{-4})$ time per character, beating the results for the exact case only when $k \gg \sqrt{m}$.

In Section 11, we describe a streaming algorithm for the approximate k-mismatch problem, which is based on our new simple approximation algorithm (Section 3), with $\tilde{O}(\varepsilon^{-2.5}\sqrt{k})$ space and $\tilde{O}(\varepsilon^{-2.5})$ time per character. In the full version [14], we introduce another sampling approach leading to an algorithm with $\tilde{O}(\varepsilon^{-2}\sqrt{k})$ space and $\tilde{O}(\varepsilon^{-3})$ time per character. Moreover, a thorough analysis of our algorithm shows that the space usage is always $\tilde{O}(\varepsilon^{-1.5}\sqrt{m})$. (Independently, Starikovskaya et al. [39] apply a different approach to design a streaming algorithm using $\tilde{O}(\varepsilon^{-2}\sqrt{m})$ space.)

2 PRELIMINARIES

For a positive integer x, let $[x] = \{0, 1, 2, \ldots, x-1\}$. A string S of length $|S| = s$ is a sequence of characters $S[0]S[1]\cdots S[s-1]$ over an alphabet Σ. In this work, we assume $\Sigma = [\sigma]$. A *substring* of S is denoted by $S[i\,..\,j) = S[i]S[i+1]\cdots S[j-1]$ for $0 \leq i \leq j \leq s$. If $i = 0$, the substring is called a *prefix* of S, and if $j = s$, the substring is called a *suffix* of S. We also denote $S[i\,..\,j)$ with $S[i\,..\,j-1]$. For two strings S and S' of the same length $|S| = s = |S'|$, we denote by $\mathsf{HD}(S, S')$ the Hamming distance of S and S', that is, $\mathsf{HD}(S, S') = |\{i \in [s] : S[i] \neq S'[i]\}|$. Let $\odot$ denote concatenation (in increasing order of the running index).

We begin with a precise statement of the problem in three variants. We state the problem in a slightly more general form, where we are additionally given a set Q of query locations. (One may take $Q = [n-m+1]$ at the end to reproduce the standard formulation.) Notice that the choice of $\frac{1}{3}$ in the problem statement is due to analysis that appears in the full version [14].

Problem 2.1. Approximate Text-to-Pattern Hamming Distances

Input: A pattern $P \in \Sigma^m$, a text $T \in \Sigma^n$, a sorted set $Q \subseteq [n-m+1]$ of query locations, and an error parameter $\varepsilon \in (0, \frac{1}{3}]$.

Output: For every $i \in Q$, a value $\widetilde{d_i}$ such that $(1-\varepsilon)d_i \leq \widetilde{d_i} \leq (1+\varepsilon)d_i$, where $d_i = \mathsf{HD}(P, T[i\,..\,i+m))$ is the Hamming distance between P and $T[i\,..\,i+m)$.

The decision version of the problem, approximately comparing each distance with a given threshold value, is formulated using the

notion of an (ε, k)-*estimation*. We say that $\tilde{x}$ is an (ε, k)-estimation of x if the following holds:

- if $\tilde{x} \in [(1-\varepsilon)k, 2(1+\varepsilon)k]$, then $(1-\varepsilon)x \le \tilde{x} \le (1+\varepsilon)x$;
- if $\tilde{x} < (1-\varepsilon)k$, then $x < k$;
- if $\tilde{x} > 2(1+\varepsilon)k$, then $x > 2k$.

Problem 2.2. Approximate Text-to-Pattern Hamming Distances with a Fixed Threshold

Input: A pattern $P \in \Sigma^m$, a text $T \in \Sigma^n$, a sorted set $Q \subseteq [n-m+1]$ of query locations, a distance threshold k, and an error parameter $\varepsilon \in (0, \frac{1}{3}]$.

Output: For every $i \in Q$, a value $\widetilde{d_i}$ that is an (ε, k)-estimation of $d_i = \mathsf{HD}(P, T[i \,..\, i+m))$.

Notice that, for every k, a solution for Problem 2.1 is also a solution for Problem 2.2. Moreover, given solutions for Problem 2.2 for each $k = 1, 2, 4, \ldots, 2^{\lfloor \log m \rfloor}$ Problem 2.1 is solved as follows: Let $\widetilde{d_i}^{(k)}$ be an (ε, k)-estimation of d_i. For every $i \in Q$ and every k, if $\widetilde{d_i}^{(k)} \in [(1-\varepsilon)k, 2(1+\varepsilon)k]$, then $\widetilde{d_i}^{(k)} \in (1 \pm \varepsilon)d_i$, and for $k = 2^{\lfloor \log d_i \rfloor}$ the condition $\widetilde{d_i}^{(k)} \in [(1-\varepsilon)k, 2(1+\varepsilon)k]$ must hold.

Local guarantees of approximation. In order to solve Problem 2.2, we first design an algorithm that solves a version which only requires a local guarantee, and is defined as follows.

Problem 2.3. Approximate Text-to-Pattern Hamming Distances with a Fixed Threshold and Local Correctness Guarantees

Input: A pattern $P \in \Sigma^m$, a text $T \in \Sigma^n$, a sorted set $Q \subseteq [n-m+1]$ of query locations, a distance threshold k, an error parameter $\varepsilon \in (0, \frac{1}{3}]$, and a real failure parameter $s > 0$.

Output: For every $i \in Q$, a value $\widetilde{d_i}$ such that the probability that $\widetilde{d_i}$ is not an (ε, k)-estimation of $d_i = \mathsf{HD}(P, T[i \,..\, i+m))$ is at most $O(1/s)$ for any fixed i.

After designing an algorithm for Problem 2.3 in Theorem 4.3, we use the new algorithm as a blackbox in the other algorithms; however, the values of s change depending on usage. Notice that, for example, if $s = O(1)$, then one can obtain an algorithm for Problem 2.2 from an algorithm for Problem 2.3 by using standard amplification techniques (see the proof of Corollary 4.4).

3 A GENERIC SAMPLING ALGORITHM

We first focus on Problem 2.3. For each position i, define $M_i := \{j \in [m] : P[j] \neq T[i+j]\}$ so that $d_i = |M_i|$. Our algorithm estimates the size d'_i of $M'_i := M_i \bmod p := \{j \bmod p : j \in M_i\}$ for an appropriately chosen integer p. By the following result, if p is a prime number picked uniformly at random from a certain range, then $(1-\varepsilon)d_i \le d'_i \le d_i$ holds with probability $1 - O(1/s)$. Thus, a good estimation of d'_i is also a good estimation for d_i.

Lemma 3.1. *Let p be a uniformly random prime in $[\hat{p}, 2\hat{p})$, where $\hat{p} = \varepsilon^{-1} sk \log m$. For every set $M \subseteq [m]$ of size $O(k)$, the probability that $|M \bmod p| < (1-\varepsilon)|M|$ is $O(1/s)$.*

Proof. The number of triples (i, j, p) such that $i, j \in M$, $i < j$, and $p \in [\hat{p}, 2\hat{p})$ is a prime divisor of $j - i$ is at most $O(|M|^2 \log_{\hat{p}} m)$ (since any positive integer in $[m]$ has at most $\log_{\hat{p}} m$ prime divisors $p \ge \hat{p}$). If $|M \bmod p| < (1-\varepsilon)|M|$, then the number of such triples with a fixed prime p is at least $\varepsilon|M|$. Thus, the number of primes $p \in [\hat{p}, 2\hat{p})$ with $|M \bmod p| < (1-\varepsilon)|M|$ is at most $O\Big(\frac{|M|^2 \log_{\hat{p}} m}{\varepsilon |M|}\Big) = O(\varepsilon^{-1} k \log m / \log \hat{p})$. The total number of primes in $[\hat{p}, 2\hat{p})$ is $\Omega(\hat{p}/\log \hat{p}) = \Omega(\varepsilon^{-1} sk \log m / \log \hat{p})$. Hence, the probability of picking a "bad" prime is $O(1/s)$. □

Offset texts and patterns. To estimate d'_i we use the concept of *offset strings* Let p be an integer. For a string S of length s and an integer $r \in [p]$, we define the rth *offset string* as

$$\bigodot_{j \in [s]:\ j \bmod p = r} S[j].$$

Notice that

$$M'_i = \left\{ r \in [p] : \bigodot_{j \in [m]:\ j \bmod p = r} P[j] \neq \bigodot_{j \in [m]:\ j \bmod p = r} T[i+j] \right\}.$$

Picking a random offset. Unfortunately, finding all occurrences of all offset patterns in all offset texts is too costly. One way to efficiently estimate d'_i is to randomly pick an offset. Let z be an integer parameter to be set later such that $1 \le z \le p$, and let $b \in [p]$ be an arbitrary integer. Write $(i \bmod p)$ as $u_i + v_i z$ with $u_i \in [z]$ and $v_i \in [\lceil p/z \rceil]$.

If the algorithm stores the offset patterns

$$\bigodot_{j \in [m]:\ (j+u) \bmod p = b} P[j]$$

for every $u \in [z]$, and the offset texts

$$\bigodot_{j \in [m]:\ (i+j-vz) \bmod p = b} T[i+j]$$

for every $v \in [\lceil p/z \rceil]$, then the algorithm has the information needed to test whether $(b - u_i) \bmod p \in M'_i$, i.e., whether

$$\bigodot_{j \in [m]:\ (j+u_i) \bmod p = b} P[j] = \bigodot_{j \in [m]:\ (j+u_i) \bmod p = b} T[i+j] = \bigodot_{j \in [m]:\ (i+j-v_i z) \bmod p = b} T[i+j].$$

Moreover, if b is chosen uniformly at random, then $(b - u_i) \bmod p$ is also uniformly random in $[p]$, and so $\Pr[(b-u_i) \bmod p \in M'_i] = \frac{d'_i}{p}$.

Picking multiple random offsets. Instead of picking one element b, our algorithm picks a random subset of elements $B \subseteq [p]$, with sampling rate $\beta = \frac{1}{2k}$. The expected size of B is small, namely, $O(\beta p) = O(\varepsilon^{-1} s \log m)$, if $p = \Theta(\varepsilon^{-1} sk \log m)$ and s is small. For each i, let E_i be the event that there exists some $b \in B$ such that $(b - u_i) \bmod p \in M'_i$.

Lemma 3.2. $\Pr[E_i] = 1 - (1-\beta)^{d'_i}$.

Proof. The set $B'_i := \{(b - u_i) \bmod p : b \in B\}$ is a subset of $[p]$ with each element sampled independently with rate β. Since E_i holds if and only if $B'_i \cap M'_i \neq \emptyset$, we have $\Pr[E_i] = 1 - (1-\beta)^{|M'_i|}$. □

Our algorithm tests for each location i whether E_i happens. This is equivalent to testing if

$$\bigodot_{j\in[m]:\ (j+u_i) \bmod p \in B} P[j] \;=\; \bigodot_{j\in[m]:\ (i+j-v_i z) \bmod p \in B} T[i+j].$$

In order to extract an estimation of d'_i, the algorithm repeats the process with $L = \varepsilon^{-2}\log s$ independent choices of B. For each location $i \in Q$, the algorithm computes c_i which is the overall number of times that the event E_i occurred throughout the L executions. Finally, the algorithm sets $\widetilde{d}_i = \log_{1-\beta}(1 - c_i/L)$ so that $c_i = (1-(1-\beta)^{\widetilde{d}_i})\cdot L$. The following pseudo-code and lemma (with proof in the full version [14]) summarize the algorithm and its correctness.

Algorithm 1: Generic-Algorithm$(T, P, Q, k, \varepsilon, s)$

```
1  Pick a random prime p ∈ [p̂, 2p̂)                          ▷ p̂ = ε^{-1} sk log m
2  p = min(p, m)
3  foreach ℓ ∈ [L] do                  ▷ L = Θ(ε^{-2} log s) sufficiently large
4      Pick a random sample B^(ℓ) ⊆ [p] with rate β = 1/(2k)
5      foreach u ∈ [z] do                                      ▷ 1 ≤ z ≤ p
6          X_u^(ℓ) = ⊙_{j∈[m]: (j+u) mod p ∈ B^(ℓ)} P[j]
7      foreach v ∈ [⌈p/z⌉] do
8          foreach i ∈ [n − m + 1] do                          ▷ Speed-up needed!
9              Y_v^(ℓ)(i) = ⊙_{j∈[m]: (i+j−vz) mod p ∈ B^(ℓ)} T[i + j]
10 foreach i ∈ Q do
11     u_i = (i mod p) mod z
12     v_i = ⌊(i mod p)/z⌋                                     ▷ (i mod p) = u_i + v_i z
13     c_i = |{ℓ ∈ [L] : X_{u_i}^(ℓ) ≠ Y_{v_i}^(ℓ)(i)}|
14     d̃_i = log_{1−β}(1 − c_i/L)
```

Lemma 3.3. *For any fixed $i \in Q$, the value $\widetilde{d}_i$ computed by Algorithm 1 is an (ε, k)-estimation of d_i with probability $1 - O(1/s)$.*

4 A SIMPLE OFFLINE IMPLEMENTATION

We now describe a simple implementation of Algorithm 1, with $O_\varepsilon(n\log^{1.5} n)$ running time. Our algorithm uses Karp–Rabin-style fingerprint functions [29, 36], summarized in the following lemma.

Lemma 4.1. (Fingerprint functions) *Given a prime number $q \ge \sigma$, define $\mathcal{F}_q = \{F_{x,q} : x \in [q]\}$, where $F_{x,q} : \Sigma^* \to [q]$ is the function $F_{x,q}(S) = (\sum_{i=0}^{|S|-1} S[i]x^i) \bmod q$. For a random $F \in \mathcal{F}_q$ and fixed distinct length-m strings X and Y, we have $\Pr[F(X) = F(Y)] \le \frac{m}{q}$.*

Proof. We have $F_{x,q}(X) = F_{x,q}(Y)$ if and only if x is one of the at most m roots of the polynomial $\sum_{i=0}^{m-1}(X[i]-Y[i])x^i$ over $\mathbb{Z}_q$. □

Our algorithm also applies the following known family of hash functions mapping $[u]$ to $\{0,1\}$:

Lemma 4.2. (Strong universal hash functions into $\{0,1\}$) *Define $\mathcal{H}_u = \{h_{x,u} : x \in [2^{\lceil \log u\rceil}]\}$, where $h_{x,u} : [u] \to \{0,1\}$ is the function with $h_{x,u}(a) = \bigoplus_{i=0}^{\ell-1} a_i x_i$, where $\oplus$ denotes exclusive-or, $\ell = \lceil \log u\rceil$, and $a_{\ell-1}\cdots a_0$ and $x_{\ell-1}\cdots x_0$ are the binary representations of a and x. For a random function $h \in \mathcal{H}_u$ and fixed numbers $a, b \in [u]$ with $a \ne b$, we have $\Pr[h(a) \ne h(b)] = \frac{1}{2}$.*

Proof. Suppose that a and b have binary representations $a_{\ell-1}\cdots a_0$ and $b_{\ell-1}\cdots b_0$. For a random $x \in [2^\ell]$ with binary representation $x_{\ell-1}\cdots x_0$, we have $h_{x,u}(a) \ne h_{x,u}(b)$ if and only if $\bigoplus_{k:\ a_k \ne b_k} x_k \ne 0$, which holds with probability exactly $\frac{1}{2}$. □

The following theorem provides a solution to Problem 2.3. We remark that the first two terms in the time bound are *sublinear* in many cases: when the threshold k is not too small (and when we choose a small s), the algorithm only needs to read a sublinear number of symbols from the text and pattern.

Theorem 4.3. *For every $s = n^{O(1)}$, there exists an algorithm for Problem 2.3 whose time cost is*

$$O\left(\sqrt{\tfrac{snm\log m}{\varepsilon^5 k}}\log s \;+\; \tfrac{n\log s}{\varepsilon^2 k} \;+\; \tfrac{1}{\varepsilon^2}|Q|\right).$$

Proof. Our solution implements Algorithm 1. Consequently, by Lemma 3.3, the algorithm solves Problem 2.3. Notice that the only changes that needed to update $Y_v^{(\ell)}(i)$ as i increments are appending $T[i+m]$ to the end and dropping $T[i]$ from the beginning. To support fast comparisons in Line 13, the strings $X_u^{(\ell)}$ and $Y_v^{(\ell)}(i)$ are not stored explicitly, but rather are represented by fingerprints $F^{(\ell)}(X_u^{(\ell)})$ and $F^{(\ell)}(Y_v^{(\ell)}(i))$ for a random function $F^{(\ell)} \in \mathcal{F}_q$, where q is some large enough prime. Thus, the comparisons in Line 13 are implemented in $O(1)$ time each; with appropriate $q = n^{O(1)}$, the comparisons are correct with probability $1 - O(1/s)$.

Analysis of running time. By a Chernoff bound, the total size of sets $B^{(\ell)}$ is $\Theta(\beta Lp)$ with probability $1 - \exp(-\Omega(\beta Lp)) = 1 - O(1/s)$ provided that the constant factor at $L = \Theta(\varepsilon^{-2}\log s)$ is sufficiently large. The analysis below is conditioned on that event. Thus, Line 4 costs $O(\beta Lp) = O(\beta Lm)$ time in total, assuming that an efficient sampling algorithm is used [12, 13].

For fixed ℓ and u, Line 6 requires computing the fingerprint of a string of length $O(\frac{m}{p}|B^{(\ell)}|)$, which costs $O(\frac{m}{p}|B^{(\ell)}|)$ time. The total cost over all $u \in [z]$ and $\ell \in [L]$ is $O(\beta mzL)$ time. For each ℓ and v, Line 9 requires computing the fingerprints of sliding windows over a string of length $O(\frac{n}{p}|B^{(\ell)}|)$, which costs $O(\frac{n}{p}|B^{(\ell)}|)$ time (as the fingerprint of a window can be computed in $O(1)$ time from the fingerprint of the previous window). The total cost over all $v \in [\lceil p/z\rceil]$ and $\ell \in [L]$ is $O(\frac{n}{p}\cdot\beta Lp\cdot\frac{p}{z}) = O(\beta n\cdot(p/z)\cdot L)$ time.

The total time bound so far is $O(\beta\cdot(mz + np/z)\cdot L)$. Setting $z = \min\left(\left\lfloor\sqrt{np/m}\right\rfloor, p\right)$ gives

$$O\left(\beta\cdot\left(\sqrt{nmp}+n\right)\cdot L\right) = O\left(\tfrac{1}{k}\left(\sqrt{nm\varepsilon^{-1}sk\log m}+n\right)\tfrac{\log s}{\varepsilon^2}\right) = O\left(\sqrt{\tfrac{snm\log m}{\varepsilon^5 k}}\log s + \tfrac{n\log s}{\varepsilon^2 k}\right).$$

In Lines 10–14, the algorithm examines the indices $i \in Q$ in increasing order. At any time, the algorithm maintains a pointer to a previous value of $F^{(\ell)}(Y_v^{(\ell)}(i))$ for each v and ℓ. As the algorithm examines the next $i \in Q$, it advances L pointers to obtain the current values of $F^{(\ell)}(Y_{v_i}^{(\ell)}(i))$ for all ℓ. Since each value $F^{(\ell)}(Y_v^{(\ell)}(i))$

changes $O(\frac{n}{p}|B^{(\ell)}|)$ times as i increases from 0 to $n-m$, the total cost for advancing the pointers is $O(\frac{n}{p}\cdot\beta Lp\cdot\frac{p}{z}) = O(\beta n\cdot(p/z)\cdot L)$, which is already accounted for. In addition, the algorithm spends $O(L)$ time per $i\in Q$, for a total of $O(|Q|L) = O(\varepsilon^{-2}|Q|\log s)$ time.

Speed-up by bit packing. We describe a simple improvement to reduce the running time of Lines 10–14 from $O(\varepsilon^{-2}|Q|\log s)$ to $O(\varepsilon^{-2}|Q|)$. Our algorithm works in the word RAM model with w-bit words, where $w=\delta\log n$ for a sufficiently small constant δ.

First, we change the fingerprint functions. At each iteration ℓ, the algorithm additionally picks a random hash function $h^{(\ell)}\in\mathcal{H}_M$ and replaces $F^{(\ell)}$ with $h^{(\ell)}\circ F^{(\ell)}$. Note that $h^{(\ell)}$ can be evaluated in $O(1)$ word operations. Let $x_u^{(\ell)} = h^{(\ell)}(F^{(\ell)}(X_u^{(\ell)}))$ and $y_v^{(\ell)}(i) = h^{(\ell)}(F^{(\ell)}(Y_v^{(\ell)}(i)))$. By Lemma 4.2, $\Pr[x_{u_i}^{(\ell)}\neq y_{v_i}^{(\ell)}(i)] = \frac{1}{2}\Pr[E_i^{(\ell)}]$ holds for each $i\in Q$. The algorithm doubles c_i to compensate.

For each $u\in[z]$, the algorithm stores $\vec{x}_u = \langle x_u^{(\ell)} : \ell\in[L]\rangle$ as a bit vector packed in $O(\lceil L/w\rceil)$ words. As i increases, the algorithm maintains the current $\vec{y}_v(i) = \langle y_v^{(\ell)}(i) : \ell\in[L]\rangle$ stored as a bit vector packed in $O(\lceil L/w\rceil)$ words for every $v\in[\lceil p/z\rceil]$. The update cost is proportional to the number of changes to $\vec{y}_v(i)$ as i increases in Lines 10–14. The number of such changes is $O(\beta n\cdot(p/z)\cdot L)$. Note that the algorithm can pre-sort the indices i at which the changes occur, for example, by a 2-pass radix sort with an $O(\sqrt{n})$-time overhead. Line 14 can then be executed by looking up the bit vectors $\vec{x}_{u_i}$ and $\vec{y}_{v_i}(i)$ and applying $O(\lceil L/w\rceil) = O(\varepsilon^{-2})$ word operations per $i\in Q$. The total time cost is $O(\varepsilon^{-2}|Q|)$.

We assumed two word operations to be available: bitwise-xor and counting the number of 1-bits in a word. If these operations are not directly supported, they can still be implemented in constant time by lookup in a table of size $2^w = n^\delta$. □

As an immediate consequence, we get the following worst-case time bound, which already improves the previous $O_\varepsilon(n\log^2 n)$ bound as a function of n.

Corollary 4.4. *There is a randomized algorithm solving Problem 2.1 in $O(\varepsilon^{-2.5}n\log^{1.5}n)$ time correctly with high probability.*

Proof. We run the algorithm for a sufficiently large constant s (in this application, the simpler version without bit packing suffices), and repeat $O(c\log n)$ times (taking the median of the answers for each $i\in Q$) to lower the error probability per $i\in Q$ to $O(n^{-c-1})$. This solves Problem 2.2 in time

$$O\left(\sqrt{\frac{nm\log m}{\varepsilon^5 k}}\log n + \frac{n\log n}{\varepsilon^2 k} + \frac{|Q|\log n}{\varepsilon^2}\right).$$

Notice that the algorithm developed in Theorem 4.3 supports processing locations $i\in Q$ online (as long as they are provided in the increasing order). Hence, we run $O(\log m)$ instances of this algorithm in parallel, one for each $k = 1,2,4,\ldots,2^{\lfloor\log m\rfloor}$.

For each $i\in Q$, the algorithm performs a binary search over the $O(\log m)$ powers of two so that i is forwarded to $O(\log\log m)$ out of the $O(\log m)$ instances of the algorithm of Theorem 4.3. The overall running time is therefore

$$O\left(\sum_k\sqrt{\frac{nm\log m}{\varepsilon^5 k}}\log n + \sum_k\frac{n\log n}{\varepsilon^2 k} + \frac{|Q|\log n\log\log m}{\varepsilon^2}\right).$$

The first two terms are geometric progressions, and so the time cost becomes

$$O\left(\sqrt{\frac{nm\log m}{\varepsilon^5}}\log n + \frac{n\log n}{\varepsilon^2} + \frac{|Q|\log n\log\log m}{\varepsilon^2}\right) = O\left(\frac{n\log^{1.5}n}{\varepsilon^{2.5}}\right).$$
□

5 OVERVIEW OF FURTHER CONSEQUENCES

Our approach leads to many further consequences, often by careful re-implementations of our generic algorithm. Here, we give a brief overview, deferring proofs to subsequent sections.

Towards a linear-time approximation algorithm. We first note that the $O_\varepsilon(n\log^{1.5}n)$ upper bound in Corollary 4.4 can be improved when approximating large distance only: from the proof, we see that the total cost of approximating distances $d_i\geq k$ is at most

$$O_\varepsilon\left(\sqrt{\frac{nm\log m}{k}}\log n + \frac{n\log n}{k} + |Q|\log n\log\log m\right) = O_\varepsilon\left(\frac{n\log^{1.5}n}{\sqrt{k}} + n\log n\log\log m\right).$$

On the other hand, for approximating small distances, e.g., $d_i\leq\log n$, we can switch to an exact algorithm, e.g., with $O(n\sqrt{k\log k})$ running time [4] (although this uses FFT). The minimum of the two yields a time bound of $O_\varepsilon(n\log n\log\log n)$ for Problem 2.1.

To do better, we combine three algorithms:

Case I: m is small, e.g., $m\leq\log^{O(1)}n$. In this case, Problem 2.3 can be solved in linear time by a simplification of our algorithm, as we show in Section 6 (see Theorem 6.1).

Case II: k is small, e.g., $k\leq m^\delta$ for some constant δ. In this case, we can switch to a known exact algorithm, e.g., one by Cole and Hariharan [19], whose running time $O(n+\frac{nk^4}{m})$ is linear for $\delta<\frac{1}{4}$. Having been designed primarily for pattern matching with respect to edit distance, Cole and Hariharan's algorithm is quite complicated and inefficient (in terms of the polynomial dependence on k). To be more self-contained, we describe an exact algorithm in Section 7 (see Theorem 7.4), which actually has a better running time of $O(n+\frac{nk^2}{m})$. (This does not require FFT.)

Case III: $k>m^\delta$ and $m>\log^{\omega(1)}n$. Here, we go back to our algorithm in Section 4, but with $s=n^{\delta/2}$, to solve Problem 2.3 (Theorem 4.3). The running time is $O_\varepsilon\left(\sqrt{\frac{snm\log m}{k}}\log s+\frac{n\log s}{k}+|Q|\right) = O_\varepsilon\left(\sqrt{snm^{1-\delta}}\log^{1.5}n+\frac{n\log n}{m^\delta}+n\right) = O_\varepsilon(n)$. The error probability $O(n^{-\delta/2})$ can be lowered by a constant number of repetitions.

In all cases, we thus obtain a linear-time approximation algorithm for Problem 2.2. The algorithm can be modified to solve Problem 2.1, though the running time increases to $O_\varepsilon(n\log\log n)$ (see Corollary 8.1). To remove the $\log\log n$ factor, we additionally use bit-packing tricks to reimplement the algorithms in all three cases. This, in fact, leads to a slightly sublinear time bound of $O\left(\frac{n\log\sigma}{\log n}+\frac{n\log^2\log n}{\varepsilon^2\log n}\right)$, proved in the full version [14] only.

Improved ε-dependence, via fast rectangular matrix multiplication. By a different implementation, it is possible to obtain $O(n\,\mathrm{polylog}\,n)$ running time without any $\varepsilon^{-O(1)}$ factor when the pattern is long enough, namely, when $m\geq\varepsilon^{-c}$ for a sufficiently large constant c.

First, we may assume that $k \geq \sqrt{m} \geq \varepsilon^{-c/2}$, for otherwise we can switch to our exact $\tilde{O}\big(n + \frac{nk^2}{m}\big)$-time algorithm.

Our algorithm in Section 4, with $s = O(1)$, has running time $\tilde{O}\left(\sqrt{\frac{mn}{\varepsilon^5 k}} + \frac{n}{\varepsilon^2 k} + \frac{1}{\varepsilon^2}|Q|\right)$. Notice that the $\varepsilon^{-O(1)}$ factors in the first two terms disappear when k is large. The third term comes from Lines 10–14, i.e., the computation of the counts c_i, which amounts to the computation of inner products between vectors $\vec{x}_u$ and $\vec{y}_v(i)$. The vectors have dimension $L = O\big(\varepsilon^{-2}\big)$. There are $O(z)$ different vectors $\vec{x}_u$, and it is not difficult to show that there are $O\big(\frac{n}{z} + \frac{n}{\varepsilon^2 k}\big)$ different vectors $\vec{y}_v(i)$ (in expectation). Therefore, this step reduces to the multiplication of an $O(z) \times O\big(\varepsilon^{-2}\big)$ matrix and an $O\big(\varepsilon^{-2}\big) \times O\big(\frac{n}{z} + \frac{n}{\varepsilon^2 k}\big)$ matrix. For k (and thus m) sufficiently large, and for an appropriate choice of z, known rectangular matrix multiplication algorithms [20] take time nearly linear in the number of output entries $\tilde{O}\big(z \cdot \big(\frac{n}{z} + \frac{n}{\varepsilon^2 k}\big)\big) = \tilde{O}(n)$. See Section 9 for the details.

Sublinear-time algorithms, via approximate nearest neighbors. When k is not too small ($k = n^{\Omega(1)}$) and the approximation ratio is constant, it is possible to obtain truly sublinear-time algorithms for finding locations with Hamming distance approximately at most k (assuming that the number of occurrences to report is sublinear).

Recall that the algorithm in Section 4 costs $\tilde{O}_\varepsilon\left(\sqrt{\frac{smn}{k}} + \frac{n}{k} + |Q|\right)$ time. Notice that the first two terms are already sublinear when k is large. Again, the third term is the bottleneck, coming from Lines 10–14, i.e., the computation of the counts c_i, which correspond to Hamming distances between vectors $\vec{x}_{u_i}$ and $\vec{y}_{v_i}(i)$. We can no longer afford to loop through all indices i, but we just want to identify all indices i for which c_i is approximately less than some threshold value. This step reduces to reporting close pairs between a set of $O(z)$ vectors and a set of $O\big(\frac{n}{z} + \frac{n}{\varepsilon^2 k}\big)$ vectors. This subproblem can be solved by using known techniques for (offline) approximate Hamming nearest neighbor search [3, 6].

Two technical issues arise. First, not all pairs of vectors should be matched (i.e., correspond to a valid index i). However, we can identify which vectors $\vec{x}_u$ to match with each $\vec{y}_v(i)$, and these vectors appear contiguously in $\vec{x}_0, \ldots, \vec{x}_{z-1}$. Second, there will be false positives—$O(\frac{n}{s})$ of them in expectation, since the error probability per position is $O(1/s)$. However, we can still choose the parameter s to keep all terms sublinear in n. See Section 10 for the details.

Streaming algorithms, via multi-stream dictionary matching. In the streaming model, we re-implement our generic algorithm differently by treating each $Y_v^{(\ell)}$ as a stream. Computing the count c_i reduces to exact matching of the pattern $X_{u_i}^{(\ell)}$ in the stream $Y_{v_i}^{(\ell)}$ for each ℓ. To this end, we could use a known streaming algorithm for pattern matching. However, because there $O(z)$ patterns $X_u^{(\ell)}$ and $O(p/z)$ streams $Y_v^{(\ell)}$ for each ℓ, we actually need a streaming pattern matching algorithm that can handle multiple patterns and multiple text streams—luckily, this variant, known as multi-stream dictionary matching, has already been addressed in a recent paper by Golan et al. [25]. The space bound is $\tilde{O}_\varepsilon(z + p/z)$, which becomes $\tilde{O}_\varepsilon\big(\sqrt{p}\big) = \tilde{O}_\varepsilon\big(\sqrt{k}\big)$ by setting $z = \sqrt{p}$, and the per-character running time is $O_\varepsilon(1)$. See Section 11 for more details and the full version [14] for an alternative streaming algorithm with improved dependence on ε in the space consumption.

6 SIMPLIFIED ALGORITHM FOR SMALL m

In this section, we note that our algorithm in Section 4 becomes quite simple if $m \leq \sqrt{n}$. As this case will be useful later, we provide a self-contained description of the simplified algorithm below:

Theorem 6.1. *For every $s = n^{O(1)}$, there is a randomized algorithm for Problem 2.3 with running time $O\big(\varepsilon^{-2}(m^2 \log s + n)\big)$.*

Proof. Our solution is presented as Algorithm 2. Compared to Algorithm 1, we set $p = m$ (the analysis involving primes becomes unnecessary!) and $z = m$ (so that the sample B is considered with all m shifts in the pattern but with just one shift in the text). Furthermore, the Karp–Rabin fingerprints are removed, with strings directly hashed to $\{0, 1\}$ using Lemma 4.2.

Algorithm 2: Simple-Algorithm$(T, P, Q, k, \varepsilon, s)$

1 **foreach** $\ell \in [L]$ **do** ▷ $L = \Theta(\varepsilon^{-2} \log s)$ sufficiently large
2 Pick a random sample $B^{(\ell)} \subseteq [m]$ with rate $\beta = \frac{1}{2k}$
3 **foreach** $i \in [m]$ **do**
4 Pick uniformly random function $h_i^{(\ell)} : [\sigma] \to \{0, 1\}$
5 **foreach** $i \in [m]$ **do**
6 $x_i^{(\ell)} = \bigoplus_{j \in [m]:\ (i+j) \bmod m \in B^{(\ell)}} h^{(\ell)}_{(i+j) \bmod m}(P[j])$
7 **foreach** $i \in [n - m + 1]$ **do**
8 $y_i^{(\ell)} = \bigoplus_{j \in [m]:\ (i+j) \bmod m \in B^{(\ell)}} h^{(\ell)}_{(i+j) \bmod m}(T[i + j])$
9 **foreach** $i \in Q$ **do**
10 $c_i = \big|\big\{\ell \in [L] : x^{(\ell)}_{i \bmod m} \neq y_i^{(\ell)}\big\}\big|$
11 $\widetilde{d_i} = \log_{1-\beta}(1 - 2c_i/L)$

Analysis of error probability. Recall that $M_i = \{j \in [m] : P[j] \neq T[i+j]\}$ is of size d_i. Define $E_i^{(\ell)}$ as the event that $(i+j) \bmod m \in B^{(\ell)}$ for some $j \in M_i$. Observe that $E_i^{(\ell)}$ holds if and only if

$$\bigodot_{j \in [m]:\ (i+j) \bmod m \in B^{(\ell)}} P[j] \neq \bigodot_{j \in [m]:\ (i+j) \bmod m \in B^{(\ell)}} T[i + j].$$

On the other hand, the construction of $B^{(\ell)}$ assures that $\Pr\big[E_i^{(\ell)}\big] = 1 - (1 - \beta)^{d_i}$ analogously to Lemma 3.2. Moreover, by Lemma 4.2, if $E_i^{(\ell)}$ holds, then $\Pr\big[x_i^{(\ell)} \neq y_i^{(\ell)}\big] = \frac{1}{2}$. Otherwise, obviously $\Pr\big[x_i^{(\ell)} \neq y_i^{(\ell)}\big] = 0$. Hence, $\Pr\big[x_i^{(\ell)} \neq y_i^{(\ell)}\big] = \frac{1}{2}\Pr\big[E_i^{(\ell)}\big] = \frac{1}{2}\big(1 - (1 - \beta)^{d_i}\big)$. Repeating the proof of Lemma 3.3 (simplified accordingly due to $p = m$), we obtain the following result:

Lemma 6.2. *For every $i \in Q$, the value $\widetilde{d_i}$ computed by Algorithm 2 is an (ε, k)-estimation of d_i with probability $1 - O(1/s)$.*

Analysis of running time. Lines 2 and 4 cost $O(m\sigma)$ time per ℓ, for a total of $O(m\sigma L) \leq O(m^2 L) = O(\varepsilon^{-2} m^2 \log s)$ time. Line 6 costs $O(m)$ time per $i \in [m]$ and ℓ, for a total of $O(m^2 L) = O(\varepsilon^{-2} m^2 \log s)$ time. Implemented using a sliding window, Line 8 costs $O(n)$ time per ℓ, for a total of $O(nL) = O(\varepsilon^{-2} n \log s)$. Line 10 costs $O(L)$ time per $i \in Q$, for a total of $O(nL)$ as well. Next, we use bit packing to speedup Lines 8 and 10.

For each $i \in [m]$, we store $\vec{x}_i = \langle x_i^{(\ell)} : \ell \in [L] \rangle$ as a bit vector packed in $O(\lceil L/w \rceil)$ words. For each $i \in [m]$ and $a \in [\sigma]$, we also store a bit vector $\vec{h}_{i,a} = \langle h_{i,a,\ell} : \ell \in [L] \rangle$, where $h_{i,a,\ell} = 0$ if $i \notin B^{(\ell)}$ and $h_{i,a,\ell} = h_i^{(\ell)}(a)$ otherwise. Then, to compute the bit vector $\vec{y}_i = \langle y_i^{(\ell)} : \ell \in [L] \rangle$ in Line 8, we can take the bitwise exclusive-or of the vectors $\vec{y}_{i-1}$, $\vec{h}_{(i-1) \bmod m,\, T[i-1]}$ and $\vec{h}_{(i+m-1) \bmod m,\, T[i+m-1]}$, in $O(\lceil L/w \rceil) = O(\varepsilon^{-2})$ time per $i \in [n - m + 1]$. The total time is $O(\varepsilon^{-2} n)$. Line 10 also costs $O(\lceil L/w \rceil) = O(\varepsilon^{-2})$ time per $i \in Q$, for a total of $O(\varepsilon^{-2}|Q|)$ time. □

The following result is obtained by combining Theorem 4.3 with the simpler (and slightly more efficient) approach for $m \ll n$.

Theorem 6.3. *For every constant $\delta > 0$, there is a randomized algorithm for Problem 2.2 with $k \geq \varepsilon^{-1} m^{\delta}$ that runs in $O(\varepsilon^{-2} n)$ time and is correct with high probability.*

Proof. If $m \leq \log^{1/\delta} n$, we run the algorithm of Theorem 6.1 with $s = n^{\delta/2}$ and $Q = [n]$, which runs in time

$$O\big(\varepsilon^{-2}\big(m^2 \log n + n\big)\big) = O\big(\varepsilon^{-2} n\big).$$

Otherwise, we run the algorithm of Theorem 4.3 with $s = n^{\delta/2}$ and $Q = [n]$, which runs in time

$$\begin{aligned} &O\left(\sqrt{\tfrac{snm \log m}{\varepsilon^5 k}} \log s + \tfrac{n \log s}{\varepsilon^2 k} + \tfrac{n}{\varepsilon^2}\right) \\ &= O\left(\varepsilon^{-2}\sqrt{n^{1+\delta/2} m^{1-\delta}} \log^{1.5} n + \tfrac{n \log n}{\varepsilon m^{\delta}} + \tfrac{n}{\varepsilon^2}\right) \\ &= O\left(\varepsilon^{-2} n^{1-\delta/4} \log^{1.5} n + \varepsilon^{-1} n + \varepsilon^{-2} n\right) = O\left(\varepsilon^{-2} n\right). \end{aligned}$$

The whole algorithm is then repeated $O(1)$ times to lower the error probability so that one can apply the union bound across $i \in Q$. □

7 EXACT ALGORITHMS

In this section, we focus on the following problem.

Problem 7.1. Exact Text-To-Pattern Hamming Distances with a Fixed Threshold

Input: A text $T \in \Sigma^n$, a pattern $P \in \Sigma^m$, and a threshold k.

Output: For each position $i \in [n - m + 1]$, compute the exact value $d_i = \mathsf{HD}(P, T[i \,..\, i + m))$ or state that $d_i > k$.

Our approach is to first use Theorem 6.3 for $\varepsilon = \frac{1}{3}$ in order to distinguish between positions i with $\widetilde{d}_i > \frac{4}{3}k$ (which can be ignored due to $d_i > k$) and positions with $\widetilde{d}_i \leq \frac{4}{3}k$ (in which case $d_i \leq 2k$ will be computed exactly). If there are few positions with $\widetilde{d}_i \leq \frac{4}{3}k$, then for each of them the *kangaroo method* (LCE queries) [33] is used to determine each d_i in $O(k)$ time after $O(n)$-time preprocessing. Otherwise, we prove that both the pattern P and the parts of the text T containing any approximate occurrence of P are approximately periodic, i.e., that there is a value $\rho = O(k)$ which is their $O(k)$-period according to the following definition:

Definition 7.2. An integer ρ is a d-period of a string X of length x if $\mathsf{HD}(X[0 \,..\, x - \rho), X[\rho \,..\, x)) \leq d$.

We first focus on a version of Problem 7.1 where P and T have an approximate period ρ given in the input. This version is studied in Section 7.1, where we prove the following result.

Theorem 7.3. *Given an integer $\rho = O(d)$, which is a d-period of both P and T, Problem 7.1 can be solved in $O\big(n + d \min\big(d, \sqrt{n \log n}\big)\big)$ time and $O(n)$ space using a randomized algorithm that returns correct answers with high probability.*

Combining Theorem 7.3 with the kangaroo method, we obtain the following result for the general case in Section 7.2.

Theorem 7.4. *There exists a randomized algorithm for Problem 7.1 that uses $O(n)$ space, costs $O\Big(n + \min\Big(\frac{nk^2}{m}, \frac{nk\sqrt{\log m}}{\sqrt{m}}\Big)\Big)$ time, and returns correct answers with high probability.*

7.1 The Case of Approximately Periodic Strings

We start by recalling a connection, originating from a classic paper by Fischer and Paterson [22], between text-to-pattern Hamming distances and the notion of a convolution of integer functions. Throughout, we only consider functions $f : \mathbb{Z} \to \mathbb{Z}$ with finite support $\mathrm{supp}(f) = \{x : f(x) \neq 0\}$, that is, such that the number of non-zero entries in f is finite. The *convolution* of two functions f and g is a function $f * g$ defined with

$$[f * g](i) = \sum_{j \in \mathbb{Z}} f(j) \cdot g(i - j).$$

For a string X and a character $a \in \Sigma$, the *characteristic function* $X_a : \mathbb{Z} \to \{0, 1\}$ is defined so that $X_a(i) = 1$ if and only if $X[i] = a$. The *cross-correlation* of strings X and Y is a function $X \otimes Y$ defined as follows, with the reverse of Y denoted by Y^R:

$$X \otimes Y = \sum_{a \in \Sigma} X_a * Y_a^R.$$

Lemma 7.5 ([22], [17, Fact 7.1]). *For every $i \in [n - m + 1]$,*

$$\mathsf{HD}(P, T[i \,..\, i + m)) = m - [T \otimes P](i + m - 1).$$

Recall that in our setting P and T have a d-period $\rho = O(d)$. For a function f and an integer ρ, the *forward difference* of f with respect to ρ is a function $\Delta_\rho[f]$ defined as $\Delta_\rho[f](i) = f(i + \rho) - f(i)$.

Observation 7.6 ([17, Observation 7.2]). *If ρ is a d-period of a string X, then $\sum_{a \in \Sigma} |\mathrm{supp}(\Delta_\rho[X_a])| \leq 2(d + \rho)$.*

To compute $T \otimes P$, one could sum up the convolutions $T_a * P_a^R$. However, the characteristic functions of T and P have total support size $\Theta(n + m)$, while the total support size of the forward differences of T and P with respect to ρ is only $O(d + \rho)$. Hence, it would be more efficient to sum up the convolutions $\Delta_\rho[T_a] * \Delta_\rho[P_a^R]$ instead. This yields the *second forward difference* of $T \otimes P$ with respect to ρ.

Lemma 7.7 (see [17, Fact 7.4]). *For strings X, Y and a positive integer ρ, we have $\Delta_\rho[\Delta_\rho[X \otimes Y]] = \sum_{a \in \Sigma} \Delta_\rho[X_a] * \Delta_\rho\big[Y_a^R\big]$.*

Note that the second forward difference $\Delta_\rho[\Delta_\rho[f]]$, denoted by $\Delta_\rho^2[f]$, satisfies $\Delta_\rho^2[f](i) = f(i+2\rho) - 2f(i+\rho) + f(i)$. Consequently, $T \otimes P$ can be retrieved using the following formula:

$$[T \otimes P](i) = \Delta_\rho^2[T \otimes P](i + 2\rho) + 2[T \otimes P](i + \rho) - [T \otimes P](i + 2\rho).$$

Since $\mathrm{supp}(T \otimes P) \subseteq [n + m - 1]$, it suffices to process subsequent indices i starting from $i = n + m - 2$ down to $i = 0$. Therefore, when computing $T \otimes P$, the values of $[T \otimes P](i + \rho)$ and $[T \otimes P](i + 2\rho)$ have already been computed in previous iterations, and so the focus is on designing a mechanism for evaluating the function $\Delta_\rho^2[T \otimes P] = \sum_{a \in \Sigma} \Delta_\rho[T_a] * \Delta_\rho[P_a^R]$.

The convolution summation problem. To design a mechanism for evaluating $\Delta_\rho^2[T \otimes P]$, we introduce a more general *convolution summation* problem stated as follows. The input is two sequences of functions $\mathcal{F} = (f_1, f_2, \ldots, f_t)$ and $\mathcal{G} = (g_1, g_2, \ldots g_t)$, and the output is the function $\mathcal{F} \otimes \mathcal{G}$ such that $[\mathcal{F} \otimes \mathcal{G}](i) = \sum_{j=1}^{t}(f_j * g_j)(i)$.

We define the *support* of a sequence of functions $\mathcal{H}$ as $\mathrm{supp}(\mathcal{H}) = \bigcup_{h\in\mathcal{H}} \mathrm{supp}(h)$. The total number of non-zero entries across $h \in \mathcal{H}$ is denoted by $\|\mathcal{H}\| = \sum_{h\in\mathcal{H}} |h|$, where $|h| = |\mathrm{supp}(h)|$.

In our setting, we assume that the input functions are given in an efficient *sparse representation* (e.g., a linked list that contains only the non-zero entries). Moreover, the output of the algorithm is restricted to the non-zero values of $\mathcal{F} \otimes \mathcal{G}$.

Lemma 7.8. *There exists a randomized algorithm that, given two sequences of functions $\mathcal{F} = (f_1, \ldots, f_t)$ and $\mathcal{G} = (g_1, \ldots, g_t)$ with non-empty supports $\mathrm{supp}(\mathcal{F}) \subseteq [n]$ and $\mathrm{supp}(\mathcal{G}) \subseteq [n]$, computes $\mathcal{F} \otimes \mathcal{G}$ (correctly with high probability) in $O(n)$ space and in time*

$$O\left(\sum_{j=1}^{t} \min(|f_j||g_j|,\ n\log n)\right) = O\left(\min\left(\|\mathcal{F}\|\|\mathcal{G}\|,\ (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{n\log n}\right)\right).$$

Proof. There are two methods that the algorithm chooses from to compute each convolution $f_j * g_j$. The first method enumerates all pairs consisting of a non-zero entry in f_j and in g_j. Using standard hashing techniques, the time cost of computing the convolution $f_j * g_j$ this way is $O(|f_j||g_j|)$. The second method of computing $f_j * g_j$ is by FFT, which costs $O(n \log n)$ time. The algorithm combines both methods by comparing $|f_j||g_j|$ to $n \log n$ for each $1 \le j \le t$ and picking the cheaper method for each particular j. Thus, the time for computing $f_j * g_j$ for any j is $O(\min(|f_j||g_j|,\ n\log n))$.

In order to reduce the space usage, the algorithm constructs $\mathcal{F} \otimes \mathcal{G}$ by iteratively computing the sum $\sum_{j=1}^{i}(f_j * g_j)$. In each iteration, the algorithm adds the function $f_j * g_j$ to the previously stored sum of functions. The summation is stored using a lookup table of size $O(\min(n, \sum_{j=1}^{t}|f_j||g_j|))$ via standard hashing techniques (notice that the exact size of the lookup table is pre-calculated). The cost of adding $f_j * g_j$ to the previous sum of functions is linear in $\mathrm{supp}(f_j * g_j)$ and thus bounded by the time cost of computing $f_j * g_j$. Hence, the total running time of the algorithm is $O\left(\sum_{j=1}^{t}\min(|f_j||g_j|,\ n\log n)\right)$.

For each j, we have $|f_j| \le \|\mathcal{F}\|$, and therefore

$$\sum_{j=1}^{t}\min(|f_j||g_j|, n\log n) \le \sum_{j=1}^{t}|f_j||g_j| \ \le\ \sum_{j=1}^{t}\|\mathcal{F}\||g_j| = \|\mathcal{F}\|\sum_{j=1}^{t}|g_j| \ =\ \|\mathcal{F}\|\|\mathcal{G}\|.$$

The second bound is obtained by recalling that $\min(x, y) \le \sqrt{xy} \le x + y$ holds for every positive x and y:

$$\sum_{j=1}^{t}\min(|f_j||g_j|, n\log n) \le \sum_{j=1}^{t}\sqrt{|f_j||g_j|n\log n} \le \sum_{j=1}^{t}(|f_j| + |g_j|)\sqrt{n\log n} = (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{n\log n}. \quad \square$$

The algorithm. We are now ready to describe and analyze the algorithm for the case of approximately periodic strings.

Theorem 7.3. *Given an integer $\rho = O(d)$, which is a d-period of both P and T, Problem 7.1 can be solved in $O\big(n + d\min\big(d, \sqrt{n\log n}\big)\big)$ time and $O(n)$ space using a randomized algorithm that returns correct answers with high probability.*

Proof. First, the algorithm constructs the forward differences $\Delta_\rho\big[P_a^R\big]$ and $\Delta_\rho[T_a]$; this costs $O(n)$ time. Let $\mathcal{F} = (\Delta_\rho[T_a])_{a\in\Sigma}$ and $\mathcal{G} = (\Delta_\rho\big[P_a^R\big])_{a\in\Sigma}$. The algorithm uses Lemma 7.8 to compute $\mathcal{F} \otimes \mathcal{G}$. Due to Observation 7.6, $\|\mathcal{F}\|, \|G\| = O(d)$, so this computation costs $O(d\min(d, \sqrt{n\log n}))$ time and, by Lemma 7.7, results in $\Delta_\rho^2[T \otimes P]$ (in a sparse representation). Finally, the algorithm retrieves $T \otimes P$ and computes the Hamming distances using Lemma 7.5. This final step costs $O(n)$ time. Overall, the running time is $O(n + d\min(d, \sqrt{n\log n}))$, and the space usage is $O(n)$. $\square$

7.2 General Case

Theorem 7.4. *There exists a randomized algorithm for Problem 7.1 that uses $O(n)$ space, costs $O\Big(n + \min\Big(\frac{nk^2}{m}, \frac{nk\sqrt{\log m}}{\sqrt{m}}\Big)\Big)$ time, and returns correct answers with high probability.*

Proof. Without loss of generality, we may assume that $k \ge \sqrt{m}$; otherwise, the stated running time is $O(n)$ anyway. Moreover, we assume that $n \le \frac{3}{2}m$; otherwise, the text T can be decomposed into parts of length at most $\frac{3}{2}m$ with overlaps of length $m - 1$, and each part of the text can be processed separately; the overall running time does not change since the dependence on n is linear.

First, the algorithm uses Theorem 6.3 with $\varepsilon = \frac{1}{3}$, which results in a sequence $\tilde{d}_i$ satisfying the following two properties with high probability: if $\tilde{d}_i > \frac{4}{3}k$, then $d_i > k$; if $\tilde{d}_i \le \frac{4}{3}k$, then $d_i \le 2k$.

Let $C = \{i \in [n-m+1] : \tilde{d}_i \le \frac{4}{3}k\}$. Observe that we may assume without loss of generality that $\min C = 0$ and $\max C = n - m$; otherwise, T can be replaced with $T[\min C\,..\,\max C + m)$ and all indices i with $d_i \le k$ are preserved (up to a shift by $\min C$).

We consider two cases depending on whether or not C contains two distinct positions at distance $\rho \le \frac{1}{2}k$ from each other. If C does not contain two such positions, then $|C| = O\big(\frac{n}{k}\big)$, and the algorithm spends $O(d_i) = O(k)$ time for each $i \in C$ to compute d_i using $1 + d_i$ *Longest Common Extension* (LCE) queries [21, 27, 33]. After $O(n + m)$-time preprocessing, these queries locate in $O(1)$ time the leftmost mismatch between any substrings of T or P. In approximate pattern matching, this technique is called the *kangaroo method*; see [4]. The overall running time is $O(n)$ in this case.

It remains to consider the case where C contains two distinct positions at distance $\rho \le \frac{1}{2}k$ from each other. We claim that in this case ρ must be an $O(k)$-period of both P and T, and so applying Theorem 7.3 with $d = O(k)$ results in the desired running time and linear space usage.

Let the positions at distance ρ be i and i' with $i < i' = i - \rho$. Due to $\mathrm{HD}(P, T[i\,..\,i + m)) \le 2k$ and $\mathrm{HD}(P, T[i'\,..\,i' + m)) \le 2k$, we conclude from the triangle inequality that:

$$\mathrm{HD}(P[0\,..\,m-\rho), P[\rho\,..\,m)) \le \mathrm{HD}(P[0\,..\,m-\rho), T[i\,..\,i+m-\rho)) + \mathrm{HD}(T[i'+\rho\,..\,i'+m), P[\rho\,..\,m)) \le 2k + 2k = 4k.$$

Hence, ρ is a $4k$-period of P. Furthermore, due to $\mathrm{HD}(P, T[0\,..\,m)) \le 2k$ (since $0 \in C$), ρ is an $8k$-period of $T[0\,..\,m)$. Similarly, ρ is an $8k$-period of $T[n-m\,..\,n)$ (since $n-m \in C$). As $n \le \frac{3}{2}m \le 2m-\rho$, these two fragments of T overlap by at least ρ characters, which implies that ρ is a $16k$-period of T. This completes the proof. □

Notice that the algorithm does not need to use FFT to achieve $O(n + \frac{nk^2}{m})$ running time (which is enough in the next section) because the weaker $O(\|\mathcal{F}\|\|\mathcal{G}\|)$ upper bound in Lemma 7.8 suffices.

8 COMBINING ALGORITHMS

In this section, we return to approximation algorithms and design an almost linear-time solution for Problem 2.1, and a linear-time solution for Problem 2.2 by combining the three algorithms from Sections 4, 6 and 7.

Corollary 8.1. *There exists a randomized algorithm for Problem 2.2 that runs in $O(\varepsilon^{-2}n)$ time and is correct with high probability. Moreover, there exists a randomized algorithm for Problem 2.1 that runs in $O(\varepsilon^{-2}n\log\log n)$ time and is correct with high probability.*

Proof. We consider three cases.

Case I: $m \le \log^2 n$. We execute the algorithm of Theorem 6.1, with s being a large enough polynomial of n, to solve Problem 2.3, and so by the union bound the same algorithm solves Problem 2.2 with high probability. The time cost is $O(\varepsilon^{-2}n)$. We solve Problem 2.1 by examining all $k \le m$ that are powers of 2, in $O(\varepsilon^{-2}n\log m) = O(\varepsilon^{-2}n\log\log n)$ time (see the discussion in Section 2).

Case II: distances $d_i \le \varepsilon^{-1}\sqrt{m}$. We run the exact algorithm of Theorem 7.4, which computes all such distances in $O(n + (\varepsilon^{-1}\sqrt{m})^2\frac{n}{m}) = O(\varepsilon^{-2}n)$ time.

Case III: distances $d_i > \varepsilon^{-1}\sqrt{m}$ for $m > \log^2 n$. We use the algorithm of Theorem 4.3 with $s = n^{0.25}$. The time cost is

$$O\left(\sqrt{\frac{snm\log m}{\varepsilon^5 k}}\log s + \frac{n\log s}{\varepsilon^2 k} + \frac{|Q|}{\varepsilon^2}\right) = O\left(\varepsilon^{-2}\sqrt{n^{1.25}m^{0.5}}\log^{1.5} n + \frac{n\log n}{\varepsilon\sqrt{m}} + \frac{|Q|}{\varepsilon^2}\right) = O(\varepsilon^{-2}n).$$

To obtain an algorithm for Problem 2.2 that is correct with high probability, we repeat the process a constant number of times.

We solve Problem 2.1 by examining all $k > \varepsilon^{-1}\sqrt{m}$ that are powers of 2 (in parallel) and performing a binary search for each $i \in Q$. The total time is

$$O\left(\sum_{k>\varepsilon^{-1}\sqrt{m}\text{ with }\log k\in\mathbb{Z}}\sqrt{\frac{snm\log m}{\varepsilon^5 k}}\log s + \frac{n\log s}{\varepsilon^2 k} + \frac{|Q|\log\log m}{\varepsilon^2}\right) = O\left(\varepsilon^{-2}\sqrt{n^{1.25}m^{0.5}}\log^{1.5} n + \frac{n\log n}{\varepsilon\sqrt{m}} + \frac{|Q|\log\log m}{\varepsilon^2}\right) = O(\varepsilon^{-2}n\log\log m).$$ □

In the full version [14], we describe further improvements using more complicated bit-packing tricks. In that setting, we assume that the input strings are *packed*, with each character stored in $\lceil\log\sigma\rceil$ bits, so that the input strings take $O(\frac{n\log\sigma}{\log n}b)$ space only. The output can be encoded in $O\left(|Q|\frac{\log\log_{1+\varepsilon} m}{\log n}\right)$ words by rounding each $\widetilde{d}_i$ down to the nearest power of $(1+\tilde{\varepsilon})$ for $\tilde{\varepsilon} = \Theta(\varepsilon)$.

Theorem 8.2. *There is a randomized algorithm for Problem 2.1 that runs in $O\left(\frac{n\log\sigma}{\log n} + \frac{n\log^2\log n}{\varepsilon^2\log n}\right)$ time and is correct with high probability.*

9 ALGORITHMS WITH IMPROVED ε-DEPENDENCE

In this section, we show that Problem 2.1 can be solved in $\tilde{O}(n)$ time without any $\varepsilon^{-O(1)}$ factors when the pattern is sufficiently long, namely, when $m > \varepsilon^{-27.22}$. For this, we combine our generic sampling algorithm of Section 3 with fast rectangular matrix multiplication [20, 35]. Specifically, we show that if an $n\times n^\alpha$ matrix and an $n^\alpha \times n$ matrix can be multiplied in $\tilde{O}(n^2)$ time, then Problem 2.1 can be solved in $\tilde{O}(n)$ time if $m > \varepsilon^{-\max(4+4/\alpha,10)}$. In particular, with $\alpha > 0.17227$ due to Coppersmith [20], the constraint reduces to $m > \varepsilon^{-27.22}$. Allowing $\hat{O}(n)$ time rather than $\tilde{O}(n)$ time, we can use a more recent result by Le Gall and Urrutia [35] with $\alpha > 0.3138$, resulting in a looser constraint $m > \varepsilon^{-16.75}$. We would like to remark, though, that these exponents 27.22 and 16.75 have not been optimized in this version of the manuscript.

We start with a solution to Problem 2.2.

Theorem 9.1. *If $k > \varepsilon^{-\max(2+2/\alpha,5)}$ and $n > \varepsilon^{-\max(4/\alpha,6)}$, then Problem 2.2 can be solved in $\tilde{O}(n)$ time using a randomized algorithm returning correct answers with high probability.*

Proof. We follow the approach of Section 3, setting $z = \min(\varepsilon^2 k, \sqrt{n})$ and a sufficiently large $s = O(1)$. As in the proof of Theorem 4.3, we map the strings $X_u^{(\ell)}$ and $Y_v^{(\ell)}(i)$ to $x_u^{(\ell)}, y_v^{(\ell)}(i) \in \{0,1\}$ using Karp–Rabin fingerprints composed with random hash functions. Let $\vec{x}_u = \langle x_u^{(\ell)} : \ell \in [L]\rangle$ and $\vec{y}_v(i) = \langle y_v^{(\ell)}(i) : \ell \in [L]\rangle$ be the vectors defined in the proof of Theorem 4.3; here, we do not pack these bit vectors, though. Recall that the vectors $\vec{x}_u$ for $u \in [z]$ can be constructed in time $O(\beta mzL) = O(\frac{mz}{\varepsilon^2 k}) = O(m)$. Similarly, the vectors $\vec{y}_v(i)$ for $v \in [\lceil p/z\rceil]$ can be maintained (for subsequent $i \in [n-m+1]$) in the overall time $O(\beta nLp/z) = \tilde{O}(\frac{n}{\varepsilon^3 z}) = \tilde{O}(\frac{n}{\varepsilon^5 k} + \frac{\sqrt{n}}{\varepsilon^3})$. Since $k > \varepsilon^{-5}$ and $n > \varepsilon^{-6}$, this time is $\tilde{O}(n)$.

It remains to implement Lines 10–14 of Algorithm 1. A naive implementation costs $\tilde{O}(\varepsilon^{-2})$ time for each i, and the bottleneck is computing c_i, which is the inner product of $\vec{x}_{u_i}$ with $\vec{y}_{v_i}(i)$; the remaining operations cost $O(1)$ time for each i. We speed up these computations by arranging *distinct* vectors $\vec{x}_{u_i}$ and $\vec{y}_{v_i}(i)$ into two matrices and multiplying the two matrices.

The number of distinct vectors $\vec{x}_{u_i}$ is at most z. The analysis for vectors $\vec{y}_{v_i}(i)$ is more involved: First, note that v_i changes $O(n/z)$ times as i increases from 0 to $n-m$. Secondly, observe that $\vec{y}_v(i)$ differs from $\vec{y}_v(i-1)$ at a given coordinate ℓ with probability $O(\beta) = O(\frac{1}{k})$. Applying a union bound, $\Pr[\vec{y}_v(i) \ne \vec{y}_v(i-1)] = O(\frac{1}{\varepsilon^2 k})$. Hence, the expected number of distinct vectors $\vec{y}_v(i)$ is $O(\frac{n}{z} + \frac{n}{\varepsilon^2 k})$. The algorithm declares a failure if this quantity exceeds the expectation by a large constant factor (the constant probability of this event adds up to the constant probability of the algorithm returning incorrect answers). Consequently, our task reduces to

multiplying two matrices of dimensions $O(z)\times O(\varepsilon^{-2})$ and $O(\varepsilon^{-2})\times O(\frac{n}{z}+\frac{n}{\varepsilon^2 k})$. Since $\frac{n}{z}\geq z$, this process costs $\tilde{O}(n+\frac{nz}{\varepsilon^2 k})=\tilde{O}(n+\frac{n\varepsilon^2 k}{\varepsilon^2 k})=\tilde{O}(n)$ time provided that $z^\alpha>\varepsilon^{-2}$, which is due to $z^\alpha=\varepsilon^{2\alpha}k^\alpha>\varepsilon^{2\alpha-2\alpha-(2\alpha)/\alpha}=\varepsilon^{-2}$ or $z^\alpha=n^{\alpha/2}>\varepsilon^{-4/\alpha\cdot\alpha/2}=\varepsilon^{-2}$.

This way, we obtained an algorithm with expected running time $\tilde{O}(n)$ and with small constant probability of error for every position $i\in Q$. We repeat the algorithm $O(\log n)$ times to achieve with high probability bounds on both correctness and running time. □

Corollary 9.2. *If $m>\varepsilon^{-\max(4+4/\alpha,10)}$, then Problem 2.1 can be solved in $\tilde{O}(n)$ time using a randomized algorithm returning correct answers with high probability.*

Proof. We apply an exact $\tilde{O}(n)$-time algorithm [16] for $k=\sqrt{m}$ (see also Theorem 7.4) to determine d_i at locations i for which $d_i\leq\sqrt{m}$. As for the distances $d_i\geq\sqrt{m}$, we apply the algorithm in Theorem 9.1 for all $\frac{1}{2}\sqrt{m}\leq k\leq m$ that are powers of two. In this setting, we have $k>\sqrt{m}>\varepsilon^{-\max(2+2/\alpha,5)}$ and $n\geq m>\varepsilon^{-\max(4+4/\alpha,10)}>\varepsilon^{-\max(4/\alpha,6)}$, so the running time of each call is $\tilde{O}(n)$, and the number of calls is $O(\log m)=\tilde{O}(1)$. □

10 SUBLINEAR-TIME ALGORITHMS

In this section, we show how to find locations with Hamming distance approximately (up to a constant factor) less than a fixed threshold value k in truly sublinear time, provided that k is not too small and the number of occurrences to report is sublinear. In comparison, all our earlier running times have an $\Omega(|Q|)$ term, which is $\Omega(n)$ in the worst case.

We use known data structures for high-dimensional approximate spherical range reporting (which is related to approximate nearest neighbor search) [6]:

Lemma 10.1. *Given a constant $c>1$, let ρ_q and ρ_u be parameters satisfying $c\sqrt{\rho_q}+(c-1)\sqrt{\rho_u}=\sqrt{2c-1}$. Let $\vec{x}_1,\ldots,\vec{x}_n$ be vectors in $\{0,1\}^d$, and let $k\in[d]$. In $\hat{O}(dn^{1+\rho_u})$ time, one can build a data structure that, given a query vector $\vec{y}\in\{0,1\}^d$ and a query interval I, reports a set A satisfying $\{i\in I:\mathrm{HD}(\vec{x}_i,\vec{y})\leq k\}\subseteq A\subseteq\{i\in I:\mathrm{HD}(\vec{x}_i,\vec{y})\leq ck\}$ with high probability, in $\hat{O}(d(n^{\rho_q}+|A|n^{\rho_u}))$ time.*

Theorem 10.2. *Given a constant $c>1$, let ρ_q and ρ_u be parameters satisfying $c\sqrt{\rho_q}+(c-1)\sqrt{\rho_u}=\sqrt{2c-1}$. Let $\varepsilon\in(0,1)$ be an arbitrarily small constant. There is a randomized algorithm that, given a text $T\in\Sigma^n$, a pattern $P\in\Sigma^m$, and an integer $k\leq m$, reports a set of locations such that every location of the text with Hamming distance at most $(1-\varepsilon)k$ is reported, and every reported location has Hamming distance at most $(1+\varepsilon)ck$. The algorithm is correct with high probability and has expected running time*

$$\hat{O}_\varepsilon\left(\begin{array}{c} n^{\frac{1+\rho_u}{2+\rho_u-\rho_q}}+\frac{n}{\left(\frac{kn}{m}\right)^{\frac{1-\rho_q}{3-3\rho_q+\rho_u}}}+\frac{n}{k^{1-\rho_q}}+\\ \\ occ\cdot\min\left(n^{\frac{\rho_u}{2+\rho_u-\rho_q}},\left(\frac{kn}{m}\right)^{\frac{\rho_u}{3-3\rho_q+\rho_u}},k^{\rho_u}\right)\end{array}\right),$$

where occ is the number of locations in the text with Hamming distance at most $(1+\varepsilon)ck$.

Proof. We follow our generic algorithm but with a few modifications. We reset $\beta=\frac{\hat{\varepsilon}}{ck}$ for some constant $\hat{\varepsilon}=\Theta(\varepsilon)$. In Lines 6 and 9, we replace the strings $X_u^{(\ell)}$ and $Y_v^{(\ell)}(i)$ with their hashed fingerprints: $x_u^{(\ell)}=h^{(\ell)}(F^{(\ell)}(X_u^{(\ell)}))$ and $y_v^{(\ell)}(i)=h^{(\ell)}(F^{(\ell)}(Y_v^{(\ell)}(i)))$ for randomly chosen functions $F^{(\ell)}\in\mathcal{F}_M$ and $h^{(\ell)}\in\mathcal{H}_M$, where $M=n^{O(1)}$ is a sufficiently large prime. We do not explicitly store $y_v^{(\ell)}(i)$ for all i, but just for those i for which $y_v^{(\ell)}(i)$ changes as i increases. As before, Lines 1–9 cost $\hat{O}_\varepsilon\left(\sqrt{\frac{snm}{k}}+\frac{n}{k}\right)$ time. However, to aim for sublinear total time, we implement Lines 10–14 differently.

Recall that $u_i\in[z]$ and $v_i\in[\lceil p/z\rceil]$ are indices defined to satisfy $(i \bmod p)=u_i+v_iz$. As i increases, if the value $y_{v_i}^{(\ell)}(i)$ changes for some $\ell\in[L]$, we say that the index i is *critical*. We reuse an argument from the proof of Theorem 9.1 to bound the number of critical indices: The index v_i changes $O(\frac{n}{z})$ times. If v_i is unchanged as i increments, then $y_{v_i}^{(\ell)}(i)$ changes only when $(i-vz) \bmod p$ or $(i+m-vz) \bmod p$ is in $B^{(\ell)}$, which happens with probability $O(\beta)=O_\varepsilon(\frac{1}{k})$. Thus, the expected number of critical indices is $O_\varepsilon(\frac{n}{z}+\frac{n}{k})$ for each fixed ℓ, and remains $\tilde{O}_\varepsilon(\frac{n}{z}+\frac{n}{k})$ over all logarithmically many $\ell\in[L]$.

We build the data structure from Lemma 10.1 storing the vectors $\vec{x}_u=\langle x_u^{(\ell)}:\ell\in[L]\rangle$ for all $u\in[z]$, in $\hat{O}(z^{1+\rho_u})$ time. Consider two consecutive critical indices a and b. For all $i\in[a,b)$, the vector $\vec{y}_{v_i}=\langle y_{v_i}^{(\ell)}:\ell\in[L]\rangle$ is unchanged. We report an index set, where every index $i\in[a,b)$ such that $\vec{x}_{u_i}$ has Hamming distance at most $\frac{1}{2}(1-(1-\beta)^k)L$ from $\vec{y}_{v_i}$ is reported, and every reported index $i\in[a,b)$ has Hamming distance at most $\frac{1}{2}(1-(1-\beta)^{(1+\tilde{\varepsilon})ck})L$, for some appropriate choice of $\tilde{\varepsilon}=\Theta(\varepsilon)$. This reduces to the type of query supported by Lemma 10.1 since $i\mapsto u_i$ maps $[a,b)$ into at most two intervals. Note that the ratio $\frac{1-(1-\beta)^{(1+\tilde{\varepsilon})ck}}{1-(1-\beta)^k}\geq\frac{(1+\tilde{\varepsilon})ck\beta-O(ck\beta)^2}{k\beta}\geq(1+\tilde{\varepsilon})c-O(\hat{\varepsilon}c^2)$ exceeds c provided that $\hat{\varepsilon}=\Theta(\tilde{\varepsilon})$ with a sufficiently small constant factor.

By a similar probabilistic analysis as before, the error probability per i is $O(\frac{1}{s})$. Thus, the total expected number of indices reported is $O(occ+\frac{n}{s})$. The time to answer all $\tilde{O}_\varepsilon(\frac{n}{z}+\frac{n}{k})$ queries using Lemma 10.1 is $\hat{O}_\varepsilon((\frac{n}{z}+\frac{n}{k})z^{\rho_q}+(occ+\frac{n}{s})z^{\rho_u})$ in expectation. The overall expected running time is

$$\hat{O}_\varepsilon\left(\sqrt{\tfrac{snm}{k}}+z^{1+\rho_u}+\left(\tfrac{n}{z}+\tfrac{n}{k}\right)z^{\rho_q}+\left(occ+\tfrac{n}{s}\right)z^{\rho_u}\right).$$

To balance all the terms, the algorithm sets $s=z^{2\rho_u/3}(\frac{kn}{m})^{1/3}$ and $z=\min\left(n^{1/(2+\rho_u-\rho_q)},(\frac{kn}{m})^{1/(3-3\rho_q+\rho_u)},k\right)$. The expected running time is bounded by the expression stated in the theorem.

The error probability per location is smaller than a constant $<\frac{1}{2}$. We can lower the error probability by repeating logarithmically many times and outputting a location when it lies in a majority of all the reported sets. □

Example 10.3. For $\rho_u=\rho_q=1/(2c-1)$ and for all $m\leq n$, the time bound is at most

$$\hat{O}_\varepsilon\left(n^{c/(2c-1)}+\frac{n}{k^{(2c-2)/(6c-5)}}+occ\cdot n^{1/(4c-2)}\right).$$

For $\rho_u=0$ and $\rho_q=(2c-1)/c^2$, the time bound is at most

$$\hat{O}_\varepsilon\left(n^{c^2/(2c^2-2c+1)}+\frac{n}{k^{1/3}}+\frac{n}{k^{(c-1)^2/c^2}}+occ\right).$$

For $c = 2$, the above bounds are $\hat{O}_\varepsilon(n^{2/3} + n/k^{2/7} + occ \cdot n^{1/6})$ and $\hat{O}_\varepsilon(n^{4/5} + n/k^{1/4} + occ)$, though other tradeoffs are possible.

For c sufficiently close to 1, one can do better by using known *offline* approximate nearest neighbor algorithms [2, 3]:

Lemma 10.4. *Let $c = 1 + \varepsilon$ for a sufficiently small constant $\varepsilon > 0$. A batch of n offline queries as in Lemma 10.1 can be answered in time*

$$O\left(d^{O(1)}\left(n^{2-\Omega(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))} + \mathcal{A}n^{O(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))}\right)\right)$$

with high probability, where $\mathcal{A}$ is the total size of the reported sets.

Theorem 10.5. *Let $c = 1+\varepsilon$ for a sufficiently small constant $\varepsilon > 0$. The expected running time in Theorem 10.2 is at most*

$$\tilde{O}\left(\frac{n}{k^{\Omega(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))}} + occ \cdot k^{O(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))}\right).$$

Proof. We proceed as in the proof of Theorem 10.2, but note that the generated queries are offline, for which Lemma 10.4 is applicable. Effectively, we can set $\rho_q = 1-\Theta(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))$ and $\rho_u = \Theta(\varepsilon^{1/3}/\log^{2/3}(1/\varepsilon))$ in the time bound. □

In the case of distinguishing between distance 0 (exact match) versus distance more than δm, the algorithm can be simplified:

Theorem 10.6. *Given a text string of length n, a pattern string of length m, and a value $\delta > 0$, there is a randomized algorithm to report a set of locations, such that every location of the text with Hamming distance 0 is reported, and every reported location has Hamming distance at most δm. The algorithm is correct with high probability and has expected running time*

$$\tilde{O}\left(\delta^{-1/3}n^{2/3} + \delta^{-1}\tfrac{n}{m} + occ\right),$$

where occ is the number of locations in the text with Hamming distance at most δm.

Proof. We proceed as in the proof of Theorem 10.2, using specific constants for c and ε (e.g., $c = 2$ and $\varepsilon = 1/3$) and setting $k = \frac{\delta m}{(1+\varepsilon)c} = \Theta(\delta m)$. We no longer need Lemma 10.1 (approximate nearest neighbor search). We use standard hashing and one-dimensional range search to find all $i \in [a, b)$ such that $\vec{x}_{u_i}$ has distance 0 from (i.e., is identical to) $\vec{y}_{v_i}$, for each pair of consecutive critical indices a and b. (The probabilistic analysis is also simplified, with no Chernoff bounds needed.) Effectively, we set $\rho_q = \rho_u = 0$ in the time bound, which becomes $\tilde{O}\left(\sqrt{n} + \frac{n^{2/3}m^{1/3}}{k^{1/3}} + \frac{n}{k} + occ\right)$. Putting $k = \Theta(\delta m)$ gives the theorem. □

Theorem 10.6 implies a sublinear-time *property tester* for pattern matching: run the algorithm for $\tilde{O}\left(\delta^{-1/3}n^{2/3} + \delta^{-1}\frac{n}{m}\right)$ steps and return "true" if the algorithm has not run to completion or at least one location has been reported. This way, if an exact match exists, then "true" is returned with high probability, and if the pattern is δ-far (i.e., has Hamming distance more than δm) from the text at every location, then "false" is returned with probability at least a constant $> \frac{1}{2}$ (which can be amplified by repetition).

Remark 10.7. There has been some past work on sublinear-time algorithms for string problems. Chang and Lawler [15] considered the exact fixed-threshold problem and described an algorithm with expected time $O(\frac{kn}{m}\log_\sigma m)$, which is sublinear when k is small (and m is not too small), but their work assumes a uniformly random text string. Andoni et al. [5] gave a sublinear-time algorithm for a *shift-finding* problem that is closely related to the approximate k-mismatch problem (their algorithm similarly uses approximate nearest neighbor search as a subroutine), but their work assumes that the pattern is a uniformly random string and the text is generated by adding random (Gaussian) noise to a shifted copy of the pattern. By contrast, our results hold for *worst-case* inputs. Truly sublinear-time algorithms have been proposed for the problem of approximating the edit distance between two strings, by Batu et al. [9] and Bar-Yossef et al. [7], but with large (polynomial) approximation factors. Bar-Yossef et al. [8] studied the "sketching complexity" of pattern matching and obtained sublinear bounds of the form $\tilde{O}(\delta^{-1}\frac{n}{m})$, but these do not correspond to actual running times. More recently, Ben-Eliezer et al. [10] gave very efficient property testers for the complementary problem of *pattern freeness* (testing whether the text is close to avoiding the pattern); their result does not imply property testers for pattern matching (in fact, they posed sublinear-time pattern matching as an open problem).

11 STREAMING ALGORITHMS

We now consider approximation algorithms in the streaming model.

Multi-stream dictionary matching. A useful building block for our algorithm is a subroutine for the multi-stream dictionary matching problem. A *dictionary* D is a set of patterns of length at most m each. In addition, there exist several streams representing different texts, and at each time step a new character arrives in one of the streams. After the arrival of a character to the ith stream, the algorithm has to report the longest pattern from D that matches a suffix of the ith text stream, or state that none of the patterns from D matches a suffix of the ith text stream. We use the algorithm of Golan et al. [25] for the multi-stream dictionary problem:

Lemma 11.1 (immediate from [25, Theorem 2]). *There exists an algorithm for the multi-stream dictionary matching problem on a dictionary D, with d patterns of length at most m each, which for t text streams costs $O(d\log m + t\log m\log d)$ words of space and $O(\log m + \log d\log\log d)$ time per character. Both these complexities are worst-case, and the algorithm is correct with high probability.*

11.1 Algorithm for Problem 2.2

In this section, we prove the following theorem.

Theorem 11.2. *There exists a streaming algorithm for Problem 2.2 where the pattern P can be preprocessed in advance and the text arrives in a stream so that $\tilde{d}_{i-m+1}$ is reported as soon as $T[i]$ arrives. The space usage of the algorithm is $\tilde{O}\left(\min\left(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m}\right)\right)$ words, the running time per character is $\tilde{O}\left(\frac{\min(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m})}{k} + \varepsilon^{-2}\right)$, and the outputs are correct with high probability.*

We implement Algorithm 1 in the streaming model with $z = \lfloor\sqrt{p}\rfloor$ (and therefore $\lceil p/z\rceil = \Theta(\sqrt{p})$) and $s = \Theta(1)$. Recall that $L = \Theta(\varepsilon^{-2}\log s)$ and $p = \Theta(\min(m, \varepsilon^{-1}sk\log m))$. For each $\ell \in [L]$, let $D^{(\ell)} = \{X_u^{(\ell)} : u \in [z]\}$ be a dictionary with z strings. For each $\ell \in [L]$ and $v \in [\lceil p/z\rceil]$, let $Y_v^{(\ell)}$ be a stream such that at time i

(after the arrival of $T[i]$), we have

$$Y_v^{(\ell)} = \bigodot_{j \le i \,:\, (j-vz) \bmod p \,\in\, B^{(\ell)}} T[j].$$

Notice that after the arrival of $T[i]$, the string $Y_v^{(\ell)}(i-m+1)$ is a suffix of $Y_v^{(\ell)}$.

Preprocessing phase. During the preprocessing phase, the algorithm chooses a random prime $p \in [\hat{p}, 2\hat{p})$ or sets $p = m$, whichever results in a smaller value of p. For each $\ell \in [L]$, the algorithm picks a random sample $B^{(\ell)} \subseteq [p]$ with rate $\beta = \frac{1}{2k}$. For each dictionary $D^{(\ell)}$, the algorithm (separately) applies the preprocessing of the multi-stream dictionary algorithm of Lemma 11.1 so that patterns from $D^{(\ell)}$ can be matched against streams $Y_v^{(\ell)}$ for $v \in [\lceil p/z \rceil]$.

Processing phase. After the arrival of $T[i]$, the algorithm appends $T[i]$ into some of the streams $Y_v^{(\ell)}$. More precisely, $T[i]$ should be inserted into $Y_v^{(\ell)}$ if and only if $(i-vz) \bmod p \in B^{(\ell)}$. A stream $Y_v^{(\ell)}$ is called *active at time i* if and only if $(i-vz) \bmod p \in B^{(\ell)}$. The following technical lemma, proved in the full version [14], is useful for efficiently retrieving the active streams at any time.

Lemma 11.3. *There exists a data structure that at any time i reports the streams active at time i. The space usage of the data structure is linear in the total number of streams, and the query time is linear in the output size (the number of active streams) with high probability.*

Using the data structure of Lemma 11.3, the algorithm retrieves all active streams and passes $T[i]$ into each of those streams. After processing $T[i]$, the dictionary matching algorithm of Lemma 11.1 identifies, for each stream $Y_v^{(\ell)}$, the current longest suffix that matches a pattern in $D^{(\ell)}$. The algorithm maintains a pointer $\pi_v^{(\ell)}$ to the longest pattern from $D^{(\ell)}$ that is a current suffix of $Y_v^{(\ell)}$, if such a pattern exists. Maintaining these pointers costs constant space per stream, and the overall time cost per text character is linear in the number of currently active streams.

Evaluating $\tilde{d}_{i-m+1}$. After updating the active streams, the algorithm estimates d_{i-m+1} by applying Lines 10–14 from Algorithm 1. In order to test whether $X^{(\ell)}_{u_{i-m+1}} = Y^{(\ell)}_{v_{i-m+1}}(i-m+1)$ the algorithm checks if $X^{(\ell)}_{u_{i-m+1}}$ is a suffix of the pattern pointed to by $\pi^{(\ell)}_{v_{i-m+1}}$.[2]

Complexity analysis. For each $\ell \in [L]$, the dictionary $D^{(\ell)}$ contains $O(\sqrt{p})$ patterns of length $O(m)$. Moreover, the number of streams $Y_v^{(\ell)}$ is also $O(\sqrt{p})$. Thus, the algorithm of Lemma 11.1 uses $O(\sqrt{p}\log m + \sqrt{p}\log m \log d) = \tilde{O}(\sqrt{p}) = \tilde{O}\big(\min\big(\sqrt{\varepsilon^{-1}k}, \sqrt{m}\big)\big)$ space. Across all $\ell \in [L]$, the space usage of all the streams is $O(|L|\sqrt{p}) = \tilde{O}\big(\min\big(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m}\big)\big)$. Since the auxiliary data structure of Lemma 11.3 takes linear space in the number of streams, the total space usage of the algorithm is $\tilde{O}\big(\min\big(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m}\big)\big)$.

As for the running time, we first bound the number of active streams at any time. For any stream $Y_v^{(\ell)}$ and any time i, the stream is active at time i if and only if $(i-vz) \bmod p \in B^{(\ell)}$, which happens with probability β independently across all the streams. By standard Chernoff bounds, the number of active streams at time i is $\tilde{O}(\beta \cdot L \cdot \lfloor p/z \rfloor) = \tilde{O}\big(1 + \frac{1}{k}\sqrt{\min(\varepsilon^{-1}k, m)}\varepsilon^{-2}\big) = \tilde{O}\Big(1 + \frac{\min(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m})}{k}\Big)$ with high probability. By a union bound over all the indices i, we have that with high probability the number of active streams is $\tilde{O}\Big(1 + \frac{\min(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m})}{k}\Big)$ at all times. For each active stream, the processing of $T[i]$ costs $\tilde{O}(1)$ time (due to Lemma 11.1). Moreover, the time cost for updating the data structure of Lemma 11.3 is linear in the number of active streams. The algorithm spends $\tilde{O}(1)$ time for each $\ell \in [L]$ to compute $c^{(\ell)}_{i-m+1}$, summing up to a total of $\tilde{O}(\varepsilon^{-2})$ time. Therefore, with high probability, the time cost per character is $\tilde{O}\Big(\frac{\min(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m})}{k} + \varepsilon^{-2}\Big)$.

By Lemma 3.3, $\hat{d}_{i-m+1}$ is an (ε, k)-estimation of d_{i-m+1} with large constant probability for each index i. In order to amplify the correctness probability, $O(\log n) = \tilde{O}(1)$ instances of the described algorithm are run in parallel, and using the standard median of means technique, the correctness probability becomes $1 - n^{-\Omega(1)}$ with just an $O(\log n)$ multiplicative overhead in the time and space complexities. Hence, Theorem 11.2 follows.

In the full version [14], we introduce another streaming algorithm which uses a different sampling method in order to improve the ε-dependence in space usage of the algorithm at the cost of degraded ε-dependence in the running time of the algorithm.

Theorem 11.4. *There exists a streaming algorithm for Problem 2.2 where the pattern P is preprocessed in advance and the text arrives in a stream so that $\tilde{d}_{i-m+1}$ is reported as soon as $T[i]$ arrives. The space usage of the algorithm is $\tilde{O}\Big(\min\Big(\frac{\sqrt{k}}{\varepsilon^2}, \frac{m}{\varepsilon\sqrt{k}}\Big)\Big)$, the time cost per character is $\tilde{O}(\varepsilon^{-3})$, and the outputs are correct with high probability.*

11.2 More General Problems

We consider the following generalization of Problem 2.1.

Problem 11.5. Approximate Text-To-Pattern Hamming Distances with a Fixed Threshold

Input: A pattern $P \in \Sigma^m$, a text $T \in \Sigma^n$, a distance threshold $k \le m$, and an error parameter $\varepsilon \in (0, \frac{1}{3}]$.

Output: For every $i \in [n-m+1]$, a value $\tilde{d}_i$ that is an (ε, k')-estimation of $d_i = \mathrm{HD}(P, T[i\,..\,i+m))$ for all $k' \le k$.

Theorem 11.6. *There exists a streaming algorithm for Problem 11.5 using $\tilde{O}\big(\min\big(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m}\big)\big)$ words of space and costing $\tilde{O}(\varepsilon^{-2})$ time per character. For every $i \in [n] \setminus [m-1]$, after the arrival of $T[i]$, the algorithm reports $\tilde{d}_{i-m+1}$ which with high probability is an (ε, k')-estimation of d_{i-m+1} for all $k' \le k$.*

Proof. Notice that Problem 2.1 is a special case of Problem 11.5 with $k = m$. The solution for Problem 11.5 is based on the solution for Problem 2.2; the reduction is similar to the reduction described in Section 2. The only difference is that we use only thresholds which are powers of 2 up to k (instead of powers of 2 up to m).

An additional speedup is obtained as follows: to cover all values of k' that are smaller than ε^{-1}, we use the exact algorithm by Clifford et al. [17], which costs $\tilde{O}\big(\sqrt{\varepsilon^{-1}}\big)$ time and uses $\tilde{O}(\varepsilon^{-1})$ words of space. Thus, the running time of the algorithm becomes

[2] The test costs constant time using standard techniques.

$$\tilde{O}\left(\sqrt{\varepsilon^{-1}} + \sum_{\varepsilon^{-1}\le k'\le k \text{ with } \log k'\in\mathbb{Z}} \frac{\min(\varepsilon^{-2.5}\sqrt{k'}, \varepsilon^{-2}\sqrt{m})}{k'} + \varepsilon^{-2}\right) = \tilde{O}\left(\varepsilon^{-0.5} + \varepsilon^{-2}\right) = \tilde{O}\left(\varepsilon^{-2}\right).$$

The space usage of the algorithm is

$$\tilde{O}\left(\varepsilon^{-1} + \sum_{\varepsilon^{-1}\le k'\le k \text{ with } \log k'\in\mathbb{Z}} \min\left(\varepsilon^{-2.5}\sqrt{k'}, \varepsilon^{-2}\sqrt{m}\right)\right) = \tilde{O}\left(\min\left(\varepsilon^{-2.5}\sqrt{k}, \varepsilon^{-2}\sqrt{m}\right)\right). \quad \square$$

In the full version [14], we adapt Theorem 11.4 accordingly:

Theorem 11.7. *There exists a streaming algorithm for Problem 11.5 using $\tilde{O}\left(\min\left(\varepsilon^{-2}\sqrt{k}, \varepsilon^{-1.5}\sqrt{m}\right)\right)$ words of space and costing $\tilde{O}(\varepsilon^{-3})$ time per character. For every $i \in [n] \setminus [m-1]$, after the arrival of $T[i]$, the algorithm reports $\tilde{d}_{i-m+1}$ which with high probability is an (ε, k')-estimation of d_{i-m+1} for any $k' \le k$.*

REFERENCES

[1] Karl R. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987. doi:10.1137/0216067.

[2] Josh Alman, Timothy M. Chan, and R. Ryan Williams. Polynomial representations of threshold functions and algorithmic applications. In *57th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2016*, pages 467–476, 2016. doi:10.1109/FOCS.2016.57.

[3] Josh Alman, Timothy M. Chan, and R. Ryan Williams. Faster deterministic and Las Vegas algorithms for offline approximate nearest neighbors in high dimensions. In *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 637–649, 2020. doi:10.1137/1.9781611975994.39.

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

[5] Alexandr Andoni, Piotr Indyk, Dina Katabi, and Haitham Hassanieh. Shift finding in sub-linear time. In *24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 457–465, 2013. doi:10.1137/1.9781611973105.33.

[6] Alexandr Andoni, Thijs Laarhoven, Ilya P. Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 47–66, 2017. doi:10.1137/1.9781611974782.4.

[7] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *45th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2004*, pages 550–559, 2004. doi:10.1109/FOCS.2004.14.

[8] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. The sketching complexity of pattern matching. In *7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2004*, volume 3122 of *LNCS*, pages 261–272. Springer, 2004. doi:10.1007/978-3-540-27821-4_24.

[9] Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 316–324, 2003. doi:10.1145/780542.780590.

[10] Omri Ben-Eliezer, Simon Korman, and Daniel Reichman. Deleting and testing forbidden patterns in multi-dimensional arrays. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 9:1–9:14, 2017. doi:10.4230/LIPIcs.ICALP.2017.9.

[11] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.

[12] Karl Bringmann and Tobias Friedrich. Exact and efficient generation of geometric random variates and random graphs. In *40th International Colloquium on Automata, Languages, and Programming, ICALP 2013*, volume 7965 of *LNCS*, pages 267–278. Springer, 2013. doi:10.1007/978-3-642-39206-1_23.

[13] Karl Bringmann and Konstantinos Panagiotou. Efficient sampling methods for discrete distributions. *Algorithmica*, 79(2):484–508, 2017. doi:10.1007/s00453-016-0205-0.

[14] Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern Hamming distances, 2020. arXiv:2001.00211.

[15] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994. doi:10.1007/BF01185431.

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

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

[18] Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming distance in a stream. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *LIPIcs*, pages 20:1–20:14, 2016. doi:10.4230/LIPIcs.ICALP.2016.20.

[19] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002. doi:10.1137/S0097539700370527.

[20] Don Coppersmith. Rapid multiplication of rectangular matrices. *SIAM Journal on Computing*, 11(3):467–471, 1982. doi:10.1137/0211037.

[21] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

[22] Michael J. Fischer and Michael S. Paterson. String matching and other products. In *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125, Providence, RI, 1974. AMS.

[23] Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986. doi:10.1145/8307.8309.

[24] Paweł Gawrychowski and Przemysław Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.

[25] Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 65:1–65:16, 2018. doi:10.4230/LIPIcs.ICALP.2018.65.

[26] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1998*, pages 166–173, 1998. doi:10.1109/SFCS.1998.743440.

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

[28] Howard J. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993. doi:10.1016/0020-0190(93)90177-B.

[29] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

[30] Tsvi Kopelowitz and Ely Porat. Breaking the variance: Approximating the Hamming distance in $1/\varepsilon$ time per alignment. In *56th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2015*, pages 601–613, 2015. doi:10.1109/FOCS.2015.43.

[31] Tsvi Kopelowitz and Ely Porat. A simple algorithm for approximating the text-to-pattern Hamming distance. In *1st Symposium on Simplicity in Algorithms, SOSA 2018*, volume 61 of *OASICS*, pages 10:1–10:5, 2018. doi:10.4230/OASIcs.SOSA.2018.10.

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

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

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

[35] Francois Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1029–1046, 2018. doi:10.1137/1.9781611975031.67.

[36] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/cbo9780511814075.

[37] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

[38] Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1996*, pages 320–328, 1996. doi:10.1109/SFCS.1996.548491.

[39] Tatiana Starikovskaya, Michal Svagerka, and Przemysław Uznański. L_p pattern matching in a stream, 2019. arXiv:1907.04405.