Hamming Distance Oracles

Itai Boneh
University of Wrocław, Poland

Dvir Fried
Bar-Ilan University, Ramat Gan, Israel

Shay Golan
Ariel University, Israel

Matan Kraus
Bar-Ilan University, Ramat Gan, Israel

Ely Porat
Bar-Ilan University, Ramat Gan, Israel

Abstract

In this paper, we present and study the *Hamming distance oracle problem*. In this problem, the task is to preprocess two strings S and T of lengths n and m, respectively, to obtain a data structure that is able to return the Hamming distance between a substring of S and a substring of T.

For strings over a constant-size alphabet, we show that for every $x \leq \min\{n, m\}$ there is a data structure with $\tilde{O}(nm/x)$ preprocessing time and $O(x)$ query time. We also provide a conditional lower bound, showing that for every $\varepsilon > 0$ there is no combinatorial data structure with query time $O(x)$ and preprocessing time $O((\frac{nm}{x})^{1-\varepsilon})$ unless combinatorial fast matrix multiplication is possible.

For strings over a general alphabet, we present a data structure with $\tilde{O}(nm/\sqrt{x})$ pre-processing time and $O(x)$ query time for every $x \leq \min\{n, m\}$. Moreover, for every $\varepsilon > 0$ we provide a data structure with a preprocessing time of $\tilde{O}(\frac{n+m}{\varepsilon^3})$ that returns with high probability a $(1 \pm \varepsilon)$ approximation of the Hamming distance of two input substrings. The query time of the approximation data structure is $\tilde{O}(1/\varepsilon^2)$.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Hamming distance, Fine-grained complexity, Data structure, Oracle

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.1

Funding *Itai Boneh*: partially supported by Israel Science Foundation grant 810/21 and by the Polish National Science Centre grant number 2023/51/B/ST6/01505.
Dvir Fried: supported by ISF grant no. 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).
Shay Golan: partially supported by Israel Science Foundation grant 810/21.
Matan Kraus: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Ely Porat: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

Acknowledgements We would like to warmly thank Panagiotis Charalampopoulos for suggesting the approximation version of the problem addressed in this paper.

1 Introduction

Given two strings S, T of the same length n, the Hamming distance $\mathsf{HD}(S, T)$ is the number of mismatches between S and T. Formally, $\mathsf{HD}(S, T) = |\{i \in [n] \mid S[i] \neq T[i]\}|$. Hamming distance is arguably the most basic and common measure of similarity between strings. The computational task of finding the Hamming distance of two given strings is trivial – it can be done in $O(n)$ time and nothing faster is possible.

In recent years, many classical string problems have been considered in the substring queries model. In this model, we are interested in preprocessing a string to obtain a data structure capable of efficiently answering queries regarding its substrings. A few examples of results in the substring queries model include finding the longest increasing subsequence of a substring [21], the longest common substring of two substrings [2], finding the period of a substring [18], and applying pattern matching [17, 19] and approximate pattern matching between substrings [7].

In this paper, we present the natural problem of constructing a *Hamming distance oracle*.

▶ **Problem 1** (Exact Hamming Distance Oracle). *Given two strings $S, T \in \Sigma^*$ of lengths n and m, respectively, construct a data structure that supports the following query: For a length ℓ and two indices $i \in [n-\ell+1]$ and $j \in [m-\ell+1]$, compute* $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$.

We also consider the approximate version of the problem. For an $\varepsilon > 0$, and $x \in \mathbb{R}$, we say that $\hat{x} \in \mathbb{R}$ is a $(1 \pm \varepsilon)$-approximation of x if $(1-\varepsilon)x \leq \hat{x} \leq (1+\varepsilon)x$.

▶ **Problem 2** (Approximate Hamming Distance Oracle). *Given two strings $S, T \in \Sigma^*$ of lengths n and m, respectively, and $\varepsilon > 0$, construct a data structure that supports the following query: For a length ℓ and two indices $i \in [n-\ell+1]$ and $j \in [m-\ell+1]$, compute a $(1 \pm \varepsilon)$-approximation of* $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$.

Our results. We begin by considering Problem 1 where the computation is in the word-RAM model. For strings over a constant-size alphabet, we show that for every $x \leq \min\{n, m\}$ there exists a data structure with $\tilde{O}(nm/x)$ preprocessing time[1] and $O(x)$ query time. We also establish a matching conditional lower bound for *combinatorial data structures*. Our upper bound, however, is non-combinatorial. This is consistent with the situation in the current state-of-the-art bounds for the classical text-to-pattern Hamming distance problem.

▶ **Theorem 3** (Informal version of Theorems 7 and 12). *For every $x \geq 0$ there exists a deterministic data structure for the Hamming distance oracle (Problem 1) for strings over a constant size alphabet with construction time $\tilde{O}(nm/x)$ and query time $O(x)$. Moreover, there is no combinatorial data structure with $O(x)$ query time with a polynomially faster construction.*

For strings over a general alphabet, i.e. polynomial integer alphabet, our techniques lead to a data structure with $\tilde{O}(nm/\sqrt{x})$ preprocessing time and $O(x)$ query time. See Figure 1 for an illustration.

For Problem 2, we show a data structure with $\tilde{O}(n)$ construction time and $\tilde{O}(1)$ query time (ignoring polynomial factors in ε^{-1}). Formally, we prove the following in Section 4.

▶ **Theorem 4.** *For every $\varepsilon > 0$, there is a randomized data structure for Problem 2 with preprocessing time $\tilde{O}(\varepsilon^{-3}(n+m))$, and query time $\tilde{O}(\varepsilon^{-2})$. The answers of the data structure are correct with high probability.*

Related work. Charalampopoulos et al. [6] considered the problem of constructing an *Edit Distance Oracle*, namely, preprocessing two input strings S and T to obtain a data structure that can compute the edit distance between any two substrings of S and T. They provided an optimal (up to sub-polynomial factors) data structure with $O(N^{1+o(1)})$ preprocessing

[1] We use $\tilde{O}$ to suppress polylogarithmic factors in the input size.

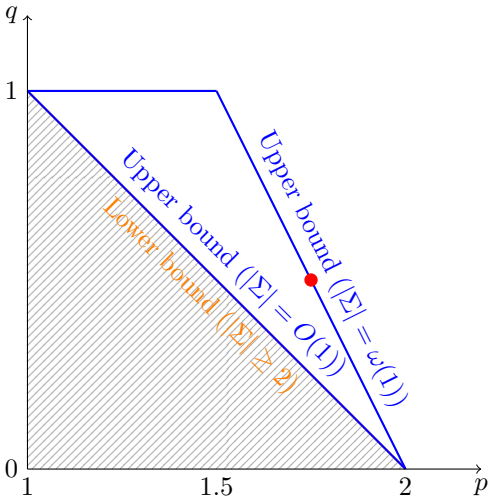

Figure 1 A summary of our results for exact oracles (for the sake of smooth presentation, the graph assumes under the assumption that $n = m$). The p-axis corresponds to the exponent of the preprocessing time and the q-axis corresponds to the exponent of the query time. For example, the point $(1.75, 0.5)$ on the slope representing the upper bound for strings over general alphabet (marked as a red dot) resembles the existence of a data structure with $\tilde{O}(n^{1.75})$ preprocessing time and $O(\sqrt{n})$ query time. The lower bound of Theorem 12 refutes any preprocessing-query trade-off that corresponds to a point in the striped area. Note that the lower bound is combinatorial.

time and $O(\mathbf{polylog}(N))$ query time, where $N = |S| \cdot |T|$. It is surprising that the problem of constructing a Hamming distance oracle has not been studied, as Hamming distance is arguably more fundamental than edit distance.

Organization. In Section 2 we present basic notations and definitions. In Section 3 we prove the stated upper and lower bounds for the (exact) Hamming distance oracles (see Theorem 7 and Theorem 12). Then, in Section 4 we address the approximation version.

2 Preliminaries

For $i, j \in \mathbb{N}$ we denote $[i..j] = \{k \in \mathbb{N} \mid i \leq k \leq j\}$. Also, $(i..j) = [i+1..j) = (i, j-1] = [i+1, j-1]$ and $[i] = [1..i]$.

A string S of length $|S| = n$ over an alphabet Σ is a sequence of characters $S = S[1]S[2]\dots S[n]$. For $i, j \in [n]$, we call $S[i..j] = S[i]S[i+1]\dots S[j]$ a *substring* of S. If $i = 1$, $S[i..j]$ is a prefix of S, and if $j = |S|$, $S[i..j]$ is a suffix of S. Let S and T be two strings over an alphabet Σ. $S \cdot T$ is the concatenation of S and T.

For two strings S and T of the same length n, the Hamming distance [14] of S and T is defined as $\mathsf{HD}(S, T) := |\{i \in [n] \mid S[i] \neq T[i]\}|$.

Text-to-Pattern Hamming distance. In the Text-to-Pattern Hamming distance problem, we are given a text $T[1..n]$ and a pattern $P[1..m]$, and we are asked to report $\mathsf{HD}(T[i..i+m), P)$ for every $i \in [1..n-m+1]$. Let $T_{\mathsf{HD}}(n, m, \Sigma)$ be the (deterministic) time complexity of computing the Text-to-Pattern Hamming Distance where n is the length of the text, m is the length of the pattern and both strings are over an alphabet Σ. Notice that for any alphabet Σ we have $T_{\mathsf{HD}}(n, m, \Sigma) = O(|\Sigma| \cdot n \log m)$ using FFT (by Fischer and Paterson [10]). For

general alphabet Σ we have $T_{\mathsf{HD}}(n, m, \Sigma) = O(n\sqrt{m \log\log m})$ (by Jin and Xu [16]). We note that if randomization is allowed, one could use the algorithm of Chan et al. [5] that solves the Text-to-Pattern Hamming distance problem in $O(n\sqrt{m})$ time.

Longest Common Prefix. The longest common prefix of S and T, denoted as $\mathsf{LCP}(S, T)$, is the maximal integer ℓ such that $S[1..\ell] = T[1..\ell]$. Our algorithms make use of the following data structure.

▶ **Lemma 5** (LCP Data structure [20, 11, 9, 15]). *There exists a data structure LCP_S that can be built for a string $S \in \Sigma^*$ of length n in $O(n)$ time. The data structure supports constant-time queries $\mathsf{LCP}_S(i, j) = \mathsf{LCP}(S[i..n], S[j..n])$.*

Predecessor, Successor, and Rank. Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence. For every $a_i \in A$, we say that i is the rank of a_i in A. For a number x, the predecessor of x in A is $\max\{a \in A \mid a \leq x\}$, and the successor of x in A is $\min\{a \in A \mid a \geq x\}$. It is known that an efficient data structure supporting rank, predecessor, and successor queries can be implemented using self-balancing search trees [1, 13].

▶ **Fact 6.** *Given a sequence A, one can construct a data structure supporting the following queries:*

1. *Given a number x, return the predecessor of x in A.*
2. *Given a number x, return the successor of x in A.*
3. *Given $a \in A$, return the rank of a in A.*

The construction time of the data structure is $\tilde{O}(n)$ and the query time is $\tilde{O}(1)$.

3 Exact Hamming Distance Oracle

In this section, we present our upper and lower bounds for Problem 1. We also conclude with a brief remark about a bounded variant of Problem 1.

3.1 Data Structure for Problem 1

Here we introduce our data structure for Problem 1 – the Hamming Distance Oracle problem.

▶ **Theorem 7.** *For every $x > 0$ there exists a deterministic data structure for Problem 1 for strings over an alphabet Σ with a preprocessing time $O(\frac{n}{x} \cdot T_{\mathsf{HD}}(m, x, \Sigma))$ and a query time of $O(x)$.*

Notice that since for every constant size alphabet Σ we have $T_{\mathsf{HD}}(m, x, \Sigma) = \tilde{O}(m)$, the upper bound of Theorem 3 follows directly from Theorem 7. For strings over general alphabet, $T_{\mathsf{HD}}(m, x, \Sigma) = \tilde{O}(m\sqrt{x})$, so Theorem 7 leads to a data structure with $\tilde{O}(nm/\sqrt{x})$ preprocessing time and $O(x)$ query time.

We first reduce Problem 1 to the problem of computing the Hamming distance of two *suffixes*. In order to present our reduction, we slightly abuse the notation and define the Hamming distance between two strings of *different* lengths as follows: Let S and T be two strings of lengths n and m respectively, we define the Hamming distance between S and T to be the Hamming distance between their prefixes of length $\min\{n, m\}$. Formally, $\mathsf{HD}(S, T) = \mathsf{HD}(S[1..\min\{n, m\}], T[1..\min\{n, m\}])$.

▶ **Problem 8** (Suffixes Hamming Distance Oracle). *Given two strings $S, T \in \Sigma^*$ of lengths n and m, respectively, construct a data structure that supports the following query: For two indices $i \in [n]$ and $j \in [m]$, compute $\mathsf{HD}(S[i..n], T[j..m])$.*

Due to the following fact, given an oracle for Problem 8, one can answer any query of Problem 1 by applying two queries to an oracle for Problem 8.

▶ **Fact 9.** *For every $i \in [n]$, $j \in [m]$ and $\ell \in [\min(n-i, m-j)]$, we have* $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell)) = \mathsf{HD}(S[i..n], T[j..m]) - \mathsf{HD}(S[i+\ell..n], T[j+\ell..m])$.

Our problem is therefore reduced to proving the following.

▶ **Lemma 10.** *Let S and T be two strings over an alphabet Σ, such that $|S| = n$, $|T| = m$ together with $x \leq m$. There exists a deterministic data structure for Problem 8 with a preprocessing time of $O(\frac{n}{x} \cdot T_{\mathsf{HD}}(m, x, \Sigma))$, and a query time of $O(\min(m, x))$.*

Proof. We first present a simple dynamic programming algorithm, which proves the theorem for $x = 1$. We then show how we can compute only a portion of the dynamic programming table, and then bound the time the data structure needs for answering a query.

Let D be a matrix of size $n \times m$ such that $\mathsf{D}[i, j] = \mathsf{HD}(S[i..n], T[j..m])$, i.e. the Hamming distance of the i-th suffix of S and the j-th suffix of T. Then, the matrix D satisfies the following recursion:

$$\mathsf{D}[i,j] = \begin{cases} \mathsf{HD}(S[i], T[j]) & \text{if } i = n \vee j = m, \\ \mathsf{HD}(S[i], T[j]) + \mathsf{D}[i+1, j+1] & \text{otherwise} \end{cases} \tag{1}$$

Notice that by Equation (1) one can compute each value $\mathsf{D}[i, j]$ in $O(1)$ time (assuming a right order of computation). Thus, filling the dynamic programming table takes $O(nm) = O(n \cdot T_{\mathsf{HD}}(m, 1, \Sigma))$ time in total[2]. Having constructed D, we have random access to the answers of all possible queries, and the data structure can answer every query in $O(1)$ time, concluding the proof of Lemma 10 for $x = 1$.

We proceed to describe our construction for any $x \geq 1$.

Preprocessing. The data structure maintains a small portion of the table D: For each $i \in [1..\lfloor \frac{n}{x} \rfloor]$, the algorithm computes and stores the $i \cdot x$-th row, i.e. for each $(i, j) \in [1..\lfloor \frac{n}{x} \rfloor] \times [m]$ the data structure computes the cell $\mathsf{D}[i \cdot x, j]$. Notice that (by Fact 9), the value of any cell $\mathsf{D}[i, j]$ is the sum of the Hamming distance $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$ and $\mathsf{D}[i+\ell, j+\ell]$ (if the indices of the cell are not in the table, we consider the cell value as 0). In particular,

$$\begin{aligned} \mathsf{D}[i \cdot x, j] &= \mathsf{HD}(S[i \cdot x..n], T[j..m]) \\ &= \mathsf{HD}(S[i \cdot x..(i+1)x), T[j..j+x)) + \mathsf{HD}(S[(i+1)x..n], T[j+x..m]) \\ &= \mathsf{HD}(S[i \cdot x..(i+1)x), T[j..j+x)) + \mathsf{D}[(i+1) \cdot x, j+x]. \end{aligned}$$

Therefore, by computing the rows in decreasing order (and using cells from rows already computed), the data structure is able to compute each row with a single invocation of a text-to-pattern Hamming distance algorithm with text T and pattern $S[i \cdot x..(i+1)x)$. In every such invocation, the text's size is $|T| = m$, the pattern size is x, and the strings are over alphabet Σ. Therefore the running time per computed row is $T_{\mathsf{HD}}(m, x, \Sigma)$. Thus, the preprocessing time is $O(\frac{n}{x} \cdot T_{\mathsf{HD}}(m, x, \Sigma))$ as required.

[2] Notice that for every alphabet Σ satisfying $|\Sigma| \geq 2$, it holds that $T_{\mathsf{HD}}(m, x, \Sigma) = \Omega(m)$ as the input must be read.

Query. Let i, j be an input query. Without loss of generality, we focus on the case $x \leq m$, since otherwise one can answer a query in $O(m) \subseteq O(x)$ time naïvely by comparing up to m pairs of characters, even without preprocessing. Let i' be the minimal integer multiple of x larger or equal to i and let $j' = j + i' - i$. The algorithm first computes $\mathsf{HD}(S[i..i'), T[j..j'))$ naïvely by comparing pairs of corresponding characters. Then the algorithm returns $\mathsf{HD}(S[i..i'), T[j..j')) + \mathsf{D}[i', j']$. Since i' is an integer multiple of x, we have that $\mathsf{D}[i', j']$ is accessible in $O(1)$ time after the preprocessing (if (i', j') is outside the boundaries of D, then either $i > n - x + 1$ or $j > m - x + 1$. In either case, the answer to the query can be naively computed in $O(x)$ time). Therefore the running time of the query is $O(i' - i + 1) = O(x)$. The correctness follows immediately from Fact 9 (see Figure 2). ◀

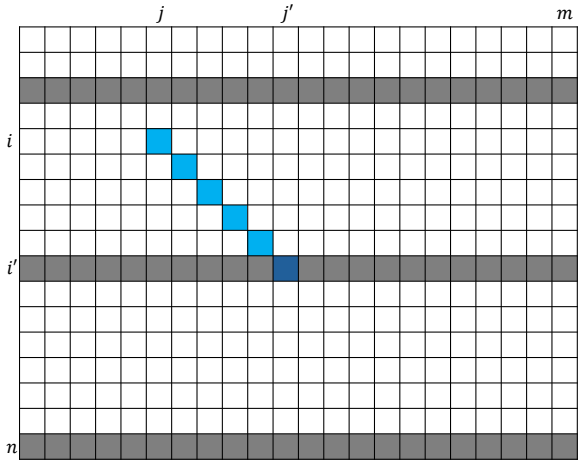

Figure 2 An example of a query. The grey rows are computed in the preprocessing. In order to compute $\mathsf{HD}(S[i..n], T[j..m])$, it is enough to compute $\mathsf{HD}(S[i..i'), T[j..j'))$ naïvely (which is the sum of the blue cells) and add $\mathsf{D}(i', j')$.

3.2 Lower Bound for Problem 1

In this section we establish the hardness of Problem 1, at least for combinatorial algorithms[3]. The lower bound is based on the known conjecture that combinatorial Boolean matrix multiplication cannot be (polynomial) faster than the naïve algorithm.

▶ **Conjecture 11** (Combinatorial Matrix Multiplication, see [12, Conjecture 12])**.** *For any $\alpha, \beta, \gamma, \varepsilon > 0$, there is no combinatorial algorithm for multiplying an $n^\alpha \times n^\beta$ matrix with an $n^\beta \times n^\gamma$ matrix in time $O((n^{\alpha+\beta+\gamma})^{(1-\varepsilon)})$* [4]*.*

Based on Conjecture 11 we establish the hardness of Problem 1 even for binary alphabet as follows.

▶ **Theorem 12** (Lower bound)**.** *Let $\mathcal{Q}$ be a combinatorial oracle for Problem 1 for strings over an alphabet Σ with $|\Sigma| \geq 2$. Let $p(n, m)$ and $q(n, m)$ be the preprocessing and query time of $\mathcal{Q}$, respectively. Then there is no $\varepsilon > 0$ such that $p(n, m) \cdot q(n, m) = O((nm)^{1-\varepsilon})$, unless Conjecture 11 is false.*

[3] The exact meaning of the term "combinatorial" is not well-defined. Nevertheless, techniques based on FFT, and more generally Boolean convolution, which are commonly used in string algorithms, are typically regarded as non-combinatorial.

[4] [12] stated the running time as $O(n^{\alpha+\beta+\gamma-\varepsilon})$. It can be easily shown that our statement is equivalent.

Proof. Our reduction is based on an idea attributed to Ely Porat and Piotr Indyk [8] that was later generalized by Gawrychowski and Uznanski [12]. Assume by contradiction that there exists some $\varepsilon > 0$ such that $p(n,m) \cdot q(n,m) = O((nm)^{1-\varepsilon})$. Let $x = q(n,m) \cdot (nm)^{\varepsilon/2}$ and note that $q(n,m) = \frac{x}{(nm)^{\varepsilon/2}}$. Let A and B be two Boolean matrices of sizes $\frac{n}{x} \times x$ and $x \times \frac{m}{x}$, respectively. The idea is to create a string S representing the rows of A and a string T representing the columns of B. We transform each row of A and each column of B such that the Hamming distance between the row's string and the column's string indicates the Boolean product of the corresponding row and column. We first show a construction using ternary alphabet, and then explain how one can convert it into a binary alphabet.

Encoding. We define encoding of values into characters, which is as follows:

- each 1 is encoded by the character $'1'$.
- each 0 from A is encoded by the character $'a'$.
- each 0 from B is encoded by the character $'b'$.

For each $i \in [\frac{n}{x}]$ the algorithm encodes the ith row of A as A_i, and for each $j \in [\frac{m}{x}]$ the algorithm encodes the jth column of B as B_j. Then S is the concatenation of A's rows (i.e. $S = A_1 \cdot A_2 \cdot ...A_{n/x}$), and T is the concatenation of B's columns ($T = B_1 \cdot B_2 \cdot ...B_{m/x}$). The resulted strings S and T have length of n and m (respectively), and then the algorithm creates the Hamming Distance Oracle data structure $\mathcal{Q}$ of S and T. We prove the following connection between Boolean matrix product and the Hamming distance of row strings and column strings.

▷ Claim 13. $(AB)_{ij} = 1 \iff \mathsf{HD}(A_i, B_j) < x$.

Proof. $(AB)_{ij} = 1$ if and only if there exists some $k \in [x]$ such that $A[i,k] = 1 = B[k,j]$. That means that both $A[i,k]$ and $B[k,j]$ were encoded by the character $'1'$, and $\mathsf{HD}(A[i,k], B[k,j]) = 0$. This implies that for A_i and B_j there is a match in at least one index, and therefore $\mathsf{HD}(A_i, B_j) < x$.

For the other direction, if $\mathsf{HD}(A_i, B_j) < x$, then for some $k \in [x]$ there is a match in $A_i[k] = B_j[k]$. Since $'a'$ does not occur in B_j and $'b'$ does not occur in A_i, it must be that $A_i[k] = 1 = B_j[k]$. Therefore, $A[i,k] = B[k,j] = 1$ and thus $(AB)_{ij} = 1$. ◁

To prove the hardness for binary alphabet one can find a binary encoding $E : \{1, a, b\} \to \{0,1\}^3$ for each symbol in $\{1, a, b\}$ that preserves the mismatches: for each $c \neq d \in \{1, a, b\}$ it holds that $\mathsf{HD}(E(c), E(d)) = 2$. Then, the reduction construction is modified to check if the distance between A_i and B_j is strictly less than $2x$. An example for such encoding is $'011', '101', '110'$.

Due to Claim 13 and the discussion above, the algorithm would query the data structure for each entry of AB if $\mathsf{HD}(A_i, B_j) < 2x$ and fill in the matrix AB accordingly.

The correctness comes directly from Claim 13. Thus, using the oracle $\mathcal{Q}$, one can compute the Boolean matrix product AB in $O(p(n,m) + \frac{nm}{x^2} q(n,m))$ time.

Recall that $p(n,m) \cdot q(n,m) = O((nm)^{1-\varepsilon})$ and $q(n,m) = \frac{x}{(nm)^{\varepsilon/2}}$. Thus, we get that

$$p(n,m) = O((nm)^{1-\varepsilon}/q(n,m)) = O((nm)^{1-\varepsilon/2}/x) = O\left(\left(\frac{nm}{x}\right)^{1-\varepsilon/2}\right).$$

Moreover,

$$\frac{nm}{x^2} q(n,m) = \frac{nm}{x^2} \frac{x}{(nm)^{\varepsilon/2}} = \frac{nm}{x} \frac{1}{(nm)^{\varepsilon/2}} = \frac{(nm)^{1-\varepsilon/2}}{x} = O\left(\left(\frac{nm}{x}\right)^{1-\varepsilon/2}\right).$$

Thus, the time for computing the product AB using the oracle $\mathcal{Q}$ is $O((\frac{nm}{x})^{1-\varepsilon/2})$, which contradicts Conjecture 11 that suggests that no algorithm can compute AB in $O((\frac{nm}{x})^{1-\varepsilon'})$ time. ◀

3.3 Remark: k-bounded Oracle

One can consider the k-bounded version of Problem 1 (for some threshold number k), where one is interested in either reporting the Hamming distance of two input substrings, or reporting that it is larger than the threshold k. We refer to this relaxed variant of a Hamming distance Oracle as a k-bounded Hamming distance oracle. The following lemma uses the method sometimes called "the Kangaroo method" [3] to construct such an oracle using near-linear time and achieves $O(k)$ query time.

▶ **Lemma 14.** *Given two strings S and T over an alphabet Σ, such that $|S| = n$, $|T| = m$ and $m \leq n$. There exists a deterministic data structure that after $O(n+m)$ preprocessing time answers k-bounded Hamming Oracle queries of any two substrings in $O(k)$ time.*

Proof. As a preprocessing, the data structure only builds the LCP data structure of Lemma 5 on $S \cdot T$ which takes $O(n+m)$ time. On a query $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$, the algorithm queries the LCP data structure to find the first mismatch between $S[i..n]$ and $T[j..m]$. If the mismatch is after $S[i+\ell)$ and $T[j+\ell)$, the algorithm reports that the Hamming distance is 0. Otherwise, the algorithm found the first mismatch $t \leq \ell$ such that $S[i+t] \neq T[j+t]$. The algorithm computes recursively $\mathsf{HD}(S[i+t+1..i+\ell), T[j+t+1..j+\ell))$ and returns the reported result with the addition of 1 (for the mismatch $S[i+t] \neq T[j+t]$). Finally, if the depth of the recursion becomes more than k, the algorithm reports that there are more than k mismatches, meaning $\mathsf{HD}(S[i..i+\ell-1], T[j..j+\ell-1]) > k$ and halts. Since every level of the recursion performs a single LCP query, the running time of the algorithm is $O(1)$ per level of the recursion, which is $O(k)$ in total. We remark that the actual implementation of the query algorithm can be done iteratively, rather than recursively, to optimize space usage and time overhead. ◀

If one is interested in a k-bounded Hamming distance oracle with $o(k)$ query time, one can simply use the data structure of Theorem 7, obtaining a similar preprocessing-query time tradeoff. One can observe that the lower bound of Theorem 12 holds for k-bounded Hamming oracles in the regime in which the query time is $o(k)$ (i.e., the queries used in the lower bound construction are between pairs of substrings with Hamming distance at most $x < k$ between them). Therefore, in the regime in which the query time is $o(k)$ and the alphabet is constant, the trade-off of Theorem 7 cannot be beaten by a combinatorial algorithm, even for a k-bounded oracle.

4 Approximate Hamming Distance Oracle

In this section, we prove Theorem 4, repeated below for easy reference.

▶ **Theorem 4.** *For every $\varepsilon > 0$, there is a randomized data structure for Problem 2 with preprocessing time $\tilde{O}(\varepsilon^{-3}(n+m))$, and query time $\tilde{O}(\varepsilon^{-2})$. The answers of the data structure are correct with high probability.*

We follow the ideas of Chan et al. [4] to obtain an oracle for $(1 \pm \varepsilon)$-approximated Hamming Distance. The algorithm of Chan et al. [4] solves the problem of approximating text-to-pattern Hamming distances. Using their techniques, with even simpler details, one can obtain the desired oracle. We provide here all the details, requiring repeating many notations and claims from [4].

We first state the definition of (ε, k)-estimate. For some integer k, we say that $\tilde{x}$ is an (ε, k)-estimate of x if the following holds:

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

We further follow [4] and introduce an integer parameter $s > 0$ controlling the probability that the algorithm returns correct answers. For a query regarding the Hamming distance of $S[i..i+\ell)$ and $T[j..j+\ell)$, define $M := \{a \in [0..\ell) : S[i+a] \neq T[j+a]\}$ so that $d := \mathsf{HD}(S[i..i+\ell), T[j..j+\ell)) = |M|$. Our algorithm estimates the size d' of $M' := M \bmod p := \{a \bmod p : a \in M\}$ for an appropriately chosen integer p. By the following lemma, if p is a prime number picked uniformly at random from a certain range, then $(1-\varepsilon)d \leq d' \leq d$ holds with probability $1 - O(1/s)$. Thus, a good estimate of d' is also a good estimate for d. The following lemma was proved in [4].

▶ **Lemma 15** ([4, Lemma 3.1]). *Let p be a random prime in $[\hat{p}..2\hat{p})$, where $\hat{p} = \varepsilon^{-1} sk \log(n+m)$. For every set $M \subseteq [m]$ of size $O(k)$, the probability that $|M \bmod p| < (1-\varepsilon)|M|$ is $O(1/s)$.*

Let p be an integer. We use the notion of offset strings. Let $\odot$ denote a concatenation. For a string S and an integer $r \in [p]$, we define the rth *offset string* of S as

$$\bigodot_{a \in [|S|):\ a \bmod p = r} S[a].$$

Notice that

$$M' = \left\{ r \in [p] \middle| \bigodot_{a \in [0,\ell):\ a \bmod p = r} S[i+a] \neq \bigodot_{a \in [0,\ell):\ a \bmod p = r} T[j+a] \right\}.$$

Picking random offsets. Our algorithm picks a random subset of elements $B \subseteq [p]$, with sampling rate $\beta = \frac{1}{2k}$. Let E be the event that there exists some $b \in B$ such that $b \in M'$. The following lemma was proved in [4].

▶ **Lemma 16** ([4, Lemma 3.2]). $\Pr[E] = 1 - (1-\beta)^{d'}$.

Our algorithm tests whether E happens. This is equivalent to testing if

$$\bigodot_{a \in [0..\ell):\ a \bmod p \in B} S[i+a] \neq \bigodot_{a \in [0..\ell):\ a \bmod p \in B} T[j+a].$$

Finally, in order to extract an estimate of d', the algorithm repeats the process with $L = \varepsilon^{-2} \log s$ independent choices of B. The algorithm computes c which is the overall number of times that the event E took place throughout the L executions. Finally, the algorithm sets $\widetilde{d} = \log_{1-\beta}(1 - c/L)$ (so that $c = (1-(1-\beta)^{\widetilde{d}}) \cdot L$). The following pseudo-code summarizes the sampling algorithm, whose correctness follows from Lemmas 15 and 16, and a standard use of Chernoff bounds. We emphasize that Line 7 only describes what mathematical objects we compute to approximate the Hamming distance. We later describe how to efficiently preprocess S and T to obtain this approximation in $\tilde{O}(1)$ time on query time.

Algorithm 1 Generic-Algorithm$(i, j, \ell, k, \varepsilon, s)$.

1 Pick a random prime $p \in [\hat{p}, 2\hat{p})$; ▷ $\hat{p} = \varepsilon^{-1} sk \log(n+m)$
2 $p = \min(p, \ell)$;
3 **foreach** $t \in [L]$ **do** ▷ $L = \Theta(\varepsilon^{-2} \log s)$ with a sufficiently large constant factor
4 Pick a random sample $B^{(t)} \subseteq [p]$ with sampling rate β; ▷ $\beta = \frac{1}{2k}$
5 Let $X^{(t)} = \bigodot_{a \in [0..\ell):\ a \bmod p \in B^{(t)}} S[i+a]$;
6 Let $Y^{(t)} = \bigodot_{a \in [0..\ell):\ a \bmod p \in B^{(t)}} T[j+a]$;
7 Set $c = |\{t \in [L] \mid X^{(t)} \neq Y^{(t)}\}|$ and $\widetilde{d} = \log_{1-\beta}(1 - c/L)$;

The following lemma was proved in [4] (with some more complicated details).

▶ **Lemma 17** ([4, Lemma 3.3]). *The value $\widetilde{d}$ computed by Line 7 is an (ε, k)-estimate of d with probability $1 - O(1/s)$.*

Implementation details

Following the discussion above, our goal is to run Line 7 in $\tilde{O}(1)$ time given a query. Notice that the computation of $\widetilde{d}$ boils down to testing if $X^{(t)} = Y^{(t)}$ exactly $L \in \tilde{O}(\varepsilon^{-2})$ times. We show how to efficiently preprocess S and T to answer this query in $\tilde{O}(1)$ time.

▶ **Lemma 18.** *Given two strings $S[1..n]$ and $T[1..m]$, an integer p, and a set $B \subseteq [0..p-1]$, there is an algorithm that constructs a (deterministic) data structure for supporting the following query: Given i, j, ℓ, report if $S_{i,\ell} = T_{j,\ell}$ with $S_{i,\ell} = \bigodot_{a \in [0..\ell): a \bmod p \in B} S[i+a]$ and $T_{j,\ell} = \bigodot_{a \in [0..\ell): a \bmod p \in B} T[j+a]$. The preprocessing time is $\tilde{O}((n+m)|B|)$ and the query time is $\tilde{O}(1)$.*

Proof. For every $r \in [1..p]$, let $I_r = (r + a \mid a \in [0..n-r] \wedge a \bmod p \in B)$ (in increasing order) and let $J_r = (r + a \mid a \in [0..m-r] \wedge a \bmod p \in B)$ (in increasing order). Let $S_r = \bigodot_{i \in I_r} S[i]$ and $T_r = \bigodot_{i \in J_r} T[i]$. Notice that for every i, ℓ there exist r, i', and i'' such that $S_{i,\ell} = S_r[i'..i'']$. Specifically, this equality holds for $r \in [1..p]$ such that $r \bmod p = i \bmod p$, i' is the rank of the successor of i in I_r and i'' is the rank of the predecessor of $i + \ell - 1$ in I_r. Similarly, for every j, ℓ there exist r', j', and j'' such that $T_{j,\ell} = T_{r'}[j'..j'']$. Specifically, this equality holds for $r' \in [1..p]$ such that $r' \bmod p = j \bmod p$, j' is the rank of the successor of j in $J_{r'}$ and j'' is the rank of the predecessor of $j + \ell - 1$ in $J_{r'}$.

As a preprocessing step, the algorithm constructs for every $r \in [1..p]$ the sequences I_r and J_r, and constructs predecessor/successor data structures for I_r and for J_r. The algorithm also constructs strings S_r and T_r, and constructs an LCP data structure over $\hat{S} = \bigodot_{r \in [1..p]} S_r \cdot T_r$. Notice that for every $r \in [1..p]$, it holds that $|S_r| \in O(n \cdot |B|/p)$ and $|T_r| \in O(m \cdot |B|/p)$. Therefore, $|\hat{S}| \in O((n+m)|B|)$. The construction time of all above data structures is therefore $\tilde{O}((n+m)|B|)$.

When a query i, j, ℓ is given, the algorithm finds r, i' and i'' such that $S_r[i'..i''] = S_{i,\ell}$ and r', j' and j'' such that $T_{r'}[j'..j''] = T_{j,\ell}$ (where finding i', i'', j' and j'' is done using predecessor and successor queries on I_r and $J_{r'}$). Since the positions of S_r and of $T_{r'}$ within $\hat{S}$ are known, the indices i', i'', j', j'' can be easily shifted to the corresponding occurrences of $S_{i,\ell}$ and of $T_{j,\ell}$ in $\hat{S}$. As we have found a representation of $S_{i,\ell}$ and of $T_{j,\ell}$ as substrings of $\hat{S}$, we can use the LCP data structure of $\hat{S}$ to check if $S_{i,\ell} = T_{j,\ell}$ in constant time. The time complexity of the query is dominated by the invocation of the predecessor/successor data structures to find i', i'', j', and j''. The running time is therefore $\tilde{O}(1)$. ◀

We now discuss how to use Lemmas 17 and 18, to preprocess S, T, ε and k and support the following query: Given i, j, ℓ return an (ε, k)-estimate of $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$. Let $\hat{p} = \varepsilon^{-1} sk \log(n+m)$, the algorithm picks a random prime $p \in [\hat{p}..2\hat{p})$. For $L = \Theta(\varepsilon^{-2} \log s)$, the data structure generates L independent samples $B^{(1)}, B^{(2)}, \ldots B^{(L)}$ with sampling rate $\beta = \frac{1}{2k}$ from $[p]$. The algorithm then applies Lemma 18 on S, T, p and $B^{(t)}$ for every $t \in [L]$. The preprocessing takes $\tilde{O}(\varepsilon^{-3}(n+m))$ expected time (since $\mathbb{E}(|B^{(t)}|) = p \cdot \beta \in O(\varepsilon^{-1} s \log(n+m))$ and $L \in \tilde{O}(\varepsilon^{-2})$). At query time, the algorithm uses the preprocessing of Lemma 18 to check if $X^{(t)} = Y^{(t)}$ for every $t \in [L]$ and compute c and $\widetilde{d}$ accordingly as in Line 7 of Line 7. It is immediate from Lemma 17 that the output is indeed an (ε, k)-estimate of $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$ with probability $1 - O(1/s)$. The running time of the query is $\tilde{O}(L) = \tilde{O}(\varepsilon^{-2})$. Choosing s to be a large enough constant implies a small constant error probability. In order to achieve (ε, k)-estimate of $\mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$ with high probability, it suffices to construct $O(c \log n)$ instances of the above data structure and return the median of the answers while processing a query.

To obtain Theorem 4, we construct the above data structure for every value of k which is a power of 2 and is at most $\min\{n, m\}$. While processing a query, the algorithm obtains an (ε, k)-estimate for each k. Let $\widetilde{d}_k$ be the estimate computed for k. If one of these estimates satisfies $\widetilde{d}_k \in [(1-\varepsilon)k..2(1+\varepsilon)k]$, the algorithm reports $\widetilde{d}_k$ as the approximation of $d := \mathsf{HD}(S[i..i+\ell), T[j..j+\ell))$. Since with high probability every $\widetilde{d}_k$ is an (ε, k)-estimate, the reported value is indeed $(1-\varepsilon)d \leq \widetilde{d} \leq (1+\varepsilon)d$. Moreover, for the (correct) value of k such that $d \in [k..2k]$, with high probability $\widetilde{d}_k \in [(1-\varepsilon)k..2(1+\varepsilon)k]$. Thus, the algorithm is correct with high probability. The construction takes[5] $\tilde{O}(\varepsilon^{-3}(n+m))$ time and the query time is $\tilde{O}(\varepsilon^{-2})$.

References

1 G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

2 Amihood Amir, Panagiotis Charalampopoulos, Solon P Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

3 Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 350–361. Springer, 2015. doi:10.1007/978-3-319-23826-5_33.

4 Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern hamming distances. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 643–656. ACM, 2020. doi:10.1145/3357713.3384266.

5 Timothy M. Chan, Ce Jin, Virginia Vassilevska Williams, and Yinzhan Xu. Faster algorithms for text-to-pattern hamming distances. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 2188–2203. IEEE, 2023. doi:10.1109/FOCS57990.2023.00136.

[5] We note that using a slightly more complicated construction, following Chan et al. [4], one can speedup the construction time to be $\tilde{O}(\varepsilon^{-2.5}(n+m))$. However, we preferred to put a simpler version here for the reader's ease.

6 Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. An almost optimal edit distance oracle. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 48:1–48:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.48.

7 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.

8 Raphaël Clifford. Matrix multiplication and pattern matching under Hamming norm. URL: https://web.archive.org/web/20160818144748/http://www.cs.bris.ac.uk/Research/Algorithms/events/BAD09/BAD09/Talks/BAD09-Hammingnotes.pdf, 2009.

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

10 Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, Massachusetts Institute of Technology, 1974.

11 Zvi Galil and Raffaele Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988. doi:10.1016/0885-064X(88)90008-8.

12 Pawel Gawrychowski and Przemyslaw Uznanski. Towards unified approximate pattern matching for hamming and l_1 distance. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 62:1–62:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.

13 L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *Information and Control*, 15(1):1–29, 1978.

14 Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

15 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

16 Ce Jin and Yinzhan Xu. Shaving logs via large sieve inequality: Faster algorithms for sparse convolution and more. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1573–1584. ACM, 2024. doi:10.1145/3618260.3649605.

17 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient data structures for the factor periodicity problem. In *String Processing and Information Retrieval: 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings 19*, pages 284–294. Springer, 2012. doi:10.1007/978-3-642-34109-0_30.

19 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53(5):1524–1577, 2024. doi:10.1137/23M1567618.

20 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.

21 Alexandre Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Math. Comput. Sci.*, 1(4):571–603, 2008. doi:10.1007/S11786-007-0033-3.