A Linear-Time $n^{0.4}$-Approximation for Longest Common Subsequence

KARL BRINGMANN, Saarland University and Max-Planck-Institute for Informatics, Saarland Informatics Campus, Germany

VINCENT COHEN-ADDAD, Sorbonne Université, UPMC Univ Paris 06, CNRS, LIP6, France

DEBARATI DAS, Basic Algorithm Research Copenhagen (BARC), University of Copenhagen, Denmark

We consider the classic problem of computing the **Longest Common Subsequence (LCS)** of two strings of length n. The 40-year-old quadratic-time dynamic programming algorithm has recently been shown to be near-optimal by Abboud, Backurs, and Vassilevska Williams [FOCS'15] and Bringmann and Künnemann [FOCS'15] assuming the Strong Exponential Time Hypothesis. This has led the community to look for sub-quadratic *approximation* algorithms for the problem.

Yet, unlike the edit distance problem for which a constant-factor approximation in almost-linear time is known, very little progress has been made on LCS, making it a notoriously difficult problem also in the realm of approximation. For the general setting, only a naive $O(n^{\varepsilon/2})$-approximation algorithm with running time $\widetilde{O}(n^{2-\varepsilon})$ has been known, for any constant $0 < \varepsilon \le 1$. Recently, a breakthrough result by Hajiaghayi, Seddighin, Seddighin, and Sun [SODA'19] provided a linear-time algorithm that yields a $O(n^{0.497956})$-approximation in expectation; improving upon the naive $O(\sqrt{n})$-approximation for the first time.

In this paper, we provide an algorithm that in time $O(n^{2-\varepsilon})$ computes an $\widetilde{O}(n^{2\varepsilon/5})$-approximation with high probability, for any $0 < \varepsilon \le 1$. Our result (1) gives an $\widetilde{O}(n^{0.4})$-approximation in linear time, improving upon the bound of Hajiaghayi, Seddighin, Seddighin, and Sun, (2) provides an algorithm whose approximation scales with any subquadratic running time $O(n^{2-\varepsilon})$, improving upon the naive bound of $O(n^{\varepsilon/2})$ for any ε, and (3) instead of only in expectation, succeeds with high probability.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; **Approximation algorithms analysis**; **Streaming, sublinear and near linear time algorithms**; *Dynamic programming;*

Additional Key Words and Phrases: Longest common subsequence, string algorithms, approximation algorithms

1 INTRODUCTION

The **longest common subsequence (LCS)** of two strings x and y is the longest string that appears as a subsequence of both strings. The length of the LCS of x and y, which we denote by $L(x, y)$, is one of the most fundamental measures of similarity between two strings and has drawn significant interest in the last five decades, see, e.g., [2–4, 6, 11, 12, 17, 20, 21, 23, 25, 27–29, 31–37]. On strings of length n, the LCS problem can be solved exactly in quadratic time $O(n^2)$ using a classical dynamic programming approach [36]. Despite an extensive line of research the quadratic running time has been improved only by logarithmic factors [31]. This lack of progress is explained by a recent result showing that any truly subquadratic algorithm for LCS would falsify the **Strong Exponential Time Hypothesis (SETH)**; this has been proven independently by Abboud et al. [2] and by Bringmann and Künnemann [20]. Further work in this direction shows that even a high polylogarithmic speedup for LCS would have surprising consequences [3, 4]. For the closely related edit distance problem the situation is similar, as the classic quadratic running time can be improved by logarithmic factors, but any truly subquadratic algorithm would falsify SETH [13].

These strong hardness results naturally bring up the question whether LCS or edit distance can be efficiently *approximated* (namely, whether an algorithm running in truly subquadratic time $O(n^{2-\varepsilon})$ for some constant $\varepsilon > 0$ can produce a *good* approximation in the worst-case). In the last two decades, significant progress has been made towards designing efficient approximation algorithms for edit distance [8, 10, 14–16, 18, 22, 24, 30]; the latest achievement is a constant-factor approximation in almost-linear[1] time [9].

For LCS the picture is much more frustrating. This problem has a simple[2] $\widetilde{O}(n^{\varepsilon/2})$-approximation algorithm with running time $O(n^{2-\varepsilon})$ for any constant $0 < \varepsilon < 1$, and it has a trivial $|\Sigma|$-approximation algorithm with running time $O(n)$ for strings over alphabet Σ. Yet, improving upon these naive bounds has evaded the community until very recently, making LCS a notoriously hard problem to approximate. In 2019, Rubinstein et al. [34] presented a subquadratic-time $O(\lambda^3)$-approximation, where λ is the ratio of the string length to the length of the optimal LCS. For strings of equal length, the approximation ratio $|\Sigma|$ was improved to $|\Sigma|-\Omega(1)$, first by Rubinstein and Song for binary alphabet [35] and then by Akmal and Vassilevska Williams for general constant-size alphabet [7]. In the general case (where λ is arbitrary and the strings can have different lengths), the naive $O(\sqrt{n})$-approximation in near-linear[3] time was recently beaten by Hajiaghayi et al. [25], who designed a linear-time algorithm that computes an $O(n^{0.497956})$-approximation *in expectation*.[4] Nonetheless, the gap between the upper bound provided by Hajiaghayi et al. [25] and the recent results on hardness of approximation [1, 5] remains huge.

1.1 Our Contribution

We present a randomized $\widetilde{O}(n^{0.4})$-approximation for LCS running in linear time $O(n)$, where the approximation guarantee holds *with high probability*.[5] More generally, we obtain a tradeoff between approximation guarantee and running time: for any $0 < \varepsilon \leq 1$ we achieve approximation ratio $\widetilde{O}(n^{2\varepsilon/5})$ in time $O(n^{2-\varepsilon})$. Formally we prove the following:

---

[1]By *almost-linear* we mean time $O(n^{1+\varepsilon})$ for a constant $\varepsilon > 0$ that can be chosen arbitrarily small.

[2]Throughout the paper $\widetilde{O}$ hides polylogarithmic factors in n.

[3]By *near-linear* we mean time $\widetilde{O}(n)$.

[4]While the SODA proceedings version of [25] claimed a high probability bound, the newer corrected Arxiv version [26] only claims that the algorithm outputs an $O(n^{0.497956})$-approximation in expectation. Personal communications with the authors confirm that the result indeed holds only in expectation. See Remarks 1 and 4 for further discussion.

[5]We say that an event happens *with high probability* (w.h.p.) if it has probability at least $1 - n^{-c}$, where the constant $c > 0$ can be chosen in advance.

THEOREM 1.1. *There is a randomized algorithm that, given strings x, y of length $n \geq 1$ and a time budget $T \in [n, n^2]$, with high probability computes a multiplicative $\widetilde{O}(n^{0.8}/T^{0.4})$-approximation of the length of the LCS of x and y in time $O(T)$.*

The improvement over the state of the art can be summarized as follows:

(1) An improved approximation ratio for the linear time regime: from $O(n^{0.497956})$ [25] to $\widetilde{O}(n^{0.4})$;
(2) A generalization to running time $O(n^{2-\varepsilon})$, breaking the naive approximation ratio $\widetilde{O}(n^{\varepsilon/2})$;
(3) The first algorithm which improves upon the naive bound *with high probability*, see Remark 1.

Remark 1 (Approximation in Expectation vs. with High Probability). Consider an algorithm with the following guarantee: with probability $1/n^\alpha$ it computes the correct LCS, and with the remaining probability it returns 0. The expected LCS returned by this algorithm is a $1/n^\alpha$ fraction of the true LCS, so this algorithm computes an n^α-approximation in expectation.

However, if we want to boost this algorithm to obtain a guarantee that holds with high probability, then we need to repeat the algorithm at least n^α times, as otherwise with probability $\Omega(1)$ all repetitions will return 0. In particular, $O(\log n)$ repetitions do not suffice. This is why an approximation in expectation is weaker than an approximation with high probability (in the context of LCS).

2 TECHNICAL OVERVIEW

Baseline Algorithm. The baseline algorithm achieves an $\widetilde{O}(n^{\varepsilon/2})$-approximation in time $O(n^{2-\varepsilon})$ for any constant $0 < \varepsilon < 1$. To this end, subsample string x by removing each symbol with probability $1 - n^{-\varepsilon/2}$. W.h.p. this reduces the length $|x|$ by a factor $n^{\varepsilon/2}$, and it reduces the length of the LCS of x and y by at most a factor $n^{\varepsilon/2}$. Then we use an algorithm that decides whether the LCS of x and y is at least L in time $\widetilde{O}(|x| \cdot L + |y|)$. For $|x| = L = n^{1-\varepsilon/2}$ we obtain running time $O(n^{2-\varepsilon})$. The details of this algorithm can be found in [25]. Specifically, [25, Algorithm 0 in Section 4] is a $\sqrt{n}$-approximation algorithm running in time $O(n \log n)$, and by changing its subsampling probability from $1 - 1/\sqrt{n}$ to $1 - n^{-\varepsilon/2}$ one obtains the baseline algorithm.

Our Approach. We combine classic exact algorithms for LCS with different subsampling strategies to develop several algorithms that work in different regimes of the problem. A combination of these algorithms then yields the full approximation algorithm. We denote the input strings by x, y.

Our Algorithm 1 covers the regime of short LCS, i.e., when the LCS has length at most n^γ for an appropriate constant $\gamma < 1$ depending on the running time budget. In this regime, we decrease the length of the string x by subsampling. This naturally allows to run classic exact algorithms for LCS on the subsampled string x (which now has significantly smaller size) and the original string y, while not deteriorating the LCS between the two strings too much.

For the remaining parts of the algorithm, the strings x and y are split into substrings $x_1, \ldots, x_{n/m}$ and $y_1, \ldots, y_{n/m}$ of length $m = n/\sqrt{T}$ where T denotes the total running time budget. For any block (i, j) we write L_{ij} for the length of the LCS of x_i and y_j. We call a set $\mathcal{S} = \{(i_1, j_1), \ldots, (i_k, j_k)\}$ with $i_1 < \ldots < i_k$ and $j_1 < \ldots < j_k$ a *block sequence*. Since we can assume the LCS of x and y to be long, it follows that there exists a good "block-aligned LCS", more precisely there exists a block sequence with large LCS sum $\sum_{(i,j)\in\mathcal{S}} L_{ij}$.

Now, a natural approach is to compute estimates $0 \leq \widetilde{L}_{ij} \leq L_{ij}$ for all blocks (i, j) and to determine the maximum sum $\widetilde{L} = \sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij}$ over all block sequences $\mathcal{S}$. Once we have estimates

$\widetilde{L}_{ij}$, the maximum $\widetilde{L}$ can be computed by dynamic programming in time $O((n/m)^2)$, which is $O(T)$ for our choice of m. In the following we describe three different strategies to compute estimates $\widetilde{L}_{ij}$. The major difficulty is that on average per block (i, j) we can only afford time $\widetilde{O}(1)$ to compute an estimate $\widetilde{L}_{ij}$.

The first strategy focuses on *matching pairs*. A matching pair of strings s, t is a pair of indices (a, b) such that the a-th symbol of s is equal to the b-th symbol of t. We write M_{ij} for the number of matching pairs of the strings x_i and y_j. Our Algorithm 2 works well if some block sequence $\mathcal{S}$ has a large total number of matching pairs $\mu = \sum_{(i,j)\in\mathcal{S}} M_{ij}$. Here the key observation (Lemma 4.5) is that for each block (i, j) there exists a symbol that occurs at least $\frac{M_{ij}}{2m}$ times in both x_i and y_j. If M_{ij} is large, matching this symbol provides a good approximation for L_{ij}. Unfortunately, since we can afford only $\widetilde{O}(1)$ running time per block, finding a frequent symbol is difficult. We develop as a new tool an algorithm that w.h.p. finds a frequent symbol in each block with an above-average number of matching pairs, see Lemma 4.6.

For our remaining two strategies we can assume the optimal LCS L to be large and μ to be small (i.e., every block sequence has a small total number of matching pairs). In our Algorithm 3, we analyze the case where $\lambda = \sum_{i,j} L_{ij}$ is large. Here we pick some diagonal and run our basic approximation algorithm on each block along the diagonal. Since there are $O(n/m)$ diagonals, an above-average diagonal has a total LCS of $\Omega(\lambda/(n/m))$. If λ is large then this provides a good estimation of the LCS. The main difficulty is how to find an above-average diagonal. A random diagonal has a good LCS sum in expectation, but not necessarily with good probability. Our solution involves non-uniform sampling, where we first test random blocks until we find a block with large LCS and then choose the diagonal containing this seed block. This sampling yields an above-average diagonal with good probability.

Recall that there always exists a block sequence $\mathcal{G}$ with large LCS sum (see Lemma 5.3). The idea of our Algorithm 4 is to focus on a uniformly random subset of all blocks, where each block is picked with probability p. Then on each picked block we can spend more time (specifically time $\widetilde{O}(1/p)$) to compute an estimate $\widetilde{L}_{ij}$. Moreover, we still find a p-fraction of $\mathcal{G}$. We analyze this algorithm in terms of μ and λ (the choice of p depends on these two parameters) and show that it works well in the complementary regimes of Algorithms 1–3.

In total we obtain four different algorithms that work well in complementary regimes of the problem. Specifically, for each setting of the parameters μ, λ, and the LCS length one of our algorithms computes a sufficiently good approximation. There are two additional complications that need to be dealt with: (1) Our algorithms need to know good guesses of μ, λ, and the LCS length in order to choose certain parameters such as subsampling rates. Since we cannot compute these parameters (note that approximating the LCS length is our task), we instead run our algorithms multiple times, using each power of two as a guess. (2) We show that for each setting of the parameters μ, λ, and the LCS length one of our algorithms computes a sufficiently good approximation and runs in time $O(T)$ – but it is not true that each of our algorithms always runs in time $O(T)$. We deal with this issue by aborting our algorithms after time $O(T)$.

Comparison with the Previous Approach of Hajiaghayi et al. [25]. The general approach of splitting x and y into blocks and performing dynamic programming over estimates $\widetilde{L}_{ij}$ was introduced by Hajiaghayi et al. [25]. Moreover, our Algorithm 1 has essentially the same guarantees as [25, Algorithm 1], but ours is a simple combination of generic parts that we reuse in our later algorithms, thus simplifying the overall procedure.

Our Algorithm 2 follows the same idea as [25, Algorithm 3], in that we want to find a frequent symbol in x_i and y_j and match only this symbol to obtain an estimate $\widetilde{L}_{ij}$. Hajiaghayi et al. find a

frequent symbol by picking a *random* symbol σ in each block x_i, y_j; in expectation σ appears at least $\frac{M_{ij}}{2m}$ times in x_i and y_j. In order to obtain a similar guarantee with high probability, we need to develop a new tool for finding frequent symbols, see Lemma 4.6 and Remark 4.

The remainder of the approach differs significantly; our Algorithms 3 and 4 are very different compared to [25, Algorithms 2 and 4]. In the following we discuss their ideas. In [25, Algorithm 2], Hajiaghayi et al. argue about the alphabet size, splitting the alphabet into frequent and infrequent letters. For infrequent letters the total number of matching pairs is small, so augmenting a classic exact algorithm by subsampling works well. Therefore, they can assume that every letter is frequent and thus the alphabet size is small. We avoid this line of reasoning. Finally, [25, Algorithm 4] is their most involved algorithm. Assuming that their other algorithms have failed to produce a sufficiently good approximation, they show that each part x_i and y_j can be turned into a *semi-permutation* by a little subsampling. Then by leveraging Dilworth's theorem and Turán's theorem they show that most blocks have an LCS length of at least $n^{1/6}$; this can be seen as a *triangle inequality* for LCS and is their most novel contribution. This results in a highly non-trivial algorithm making clever use of combinatorial machinery.

We show that these ideas can be completely avoided, by instead relying on classic algorithms based on matching pairs augmented by subsampling. Specifically, we replace their combinatorial machinery by our Algorithms 3 and 4 discussed earlier (recall that Algorithm 3 considers a non-uniformly sampled random diagonal while Algorithm 4 subsamples the set of blocks to be able to spend more time per block). We stress that our solution completely avoids the concept of semi-permutation or any sophisticated combinatorial tools as used in [25, Algorithm 4], while providing a significantly improved approximation guarantee.

Organization of the Paper. Section 3 introduces notation and a classical algorithm by Hunt and Szymanski. In Section 4 we present our new tools, in particular for finding frequent symbols. Section 5 contains our main algorithm, split into four parts that are presented in Sections 5.1, 5.3, 5.4, and 5.5, and combined in Section 5.6. More detailed pseudocode for all our algorithms can be found in the Arxiv version of this paper.

3 PRELIMINARIES

For $n \in \mathbb{N}$ we write $[n] = \{1, 2, \ldots, n\}$. By the notation $\widetilde{O}$ and $\widetilde{\Omega}$ we hide factors of the form $\text{polylog}(n)$. We use **"with high probability" (w.h.p.)** to denote probabilities of the form $1 - n^{-c}$, where the constant $c > 0$ can be chosen in advance.

String Notation. A string x over alphabet Σ is a finite sequence of letters in Σ. We denote its length by $|x|$ and its i-th letter by $x[i]$. We also denote by $x[i..j]$ the substring consisting of letters $x[i] \ldots x[j]$. For any indices $i_1 < i_2 < \ldots < i_k$ the string $z = x[i_1] \ldots x[i_k]$ forms a *subsequence* of x. For strings x, y we denote by $L(x, y)$ the length of the longest common subsequence of x and y. In this paper we study the problem of approximating $L(x, y)$ for given strings x, y of length n. While we focus on the length $L(x, y)$, our algorithms can be easily adapted to also reconstruct a subsequence attaining the output length. If x, y are clear from context, we may replace $L(x, y)$ by L. Throughout the paper we assume that the alphabet is $\Sigma \subseteq [O(n)]$ (this is without loss of generality after a $\widetilde{O}(n)$-time preprocessing).

Matching Pairs. For a symbol $\sigma \in \Sigma$, we denote the number of times that σ appears in x by $\#_\sigma(x)$, and call this the *frequency* of σ in x. For strings x and y, a *matching pair* is a pair (i, j) with $x[i] = y[j]$. We denote the number of matching pairs by $M(x, y)$. If x, y are clear from the context, we may replace $M(x, y)$ by M. Observe that $M = \sum_{\sigma \in \Sigma} \#_\sigma(x) \cdot \#_\sigma(y)$. Using this equation we can compute M in time $O(n)$.

Hunt and Szymanski [28] solved the LCS problem in time $\widetilde{O}(n + M)$. More precisely, their algorithm can be viewed as having a preprocessing phase that only reads y and runs in time $\widetilde{O}(|y|)$, and a query phase that reads x and y and takes time $\widetilde{O}(|x| + M)$. For convenience, we provide a proof sketch of their theorem in Appendix A.

THEOREM 3.1 (HUNT AND SZYMANSKI [28]). *We can preprocess a string y in time $\widetilde{O}(|y|)$. Given a string x and a preprocessed string y, we can compute their LCS in time $\widetilde{O}(|x| + M)$.*

Chernoff Bound. We frequently use the following standard variant of the Chernoff bound.

THEOREM 3.2 (MULTIPLICATIVE CHERNOFF). *Let $X_1, \ldots, X_n$ be independent random variables taking values in $\{0, 1\}$, and let $X := X_1 + \ldots + X_n$. Then we have $\Pr[X < \mathbb{E}[X]/2] \le \exp(-\mathbb{E}[X]/8)$.*

4 NEW BASIC TOOLS

4.1 Basic Approximation Algorithm

Throughout this section we abbreviate $L = L(x, y)$ and $M = M(x, y)$. We start with the basic approximation algorithm that is central to our approach; most of our later algorithms use this as a subroutine. This algorithm subsamples the string x and then runs Hunt and Szymanski's algorithm (Theorem 3.1).

LEMMA 4.1 (BASIC APPROXIMATION ALGORITHM). *Let $x, y \in \Sigma^n$. We can preprocess y in time $\widetilde{O}(n)$. Given x, the preprocessed string y, and $\beta \ge 1$, in expected time $\widetilde{O}((n + M)/\beta + 1)$ we can compute a value $\widetilde{L} \le L$ that w.h.p. satisfies $\widetilde{L} > \frac{L}{\beta} - 1$.*

PROOF. In the preprocessing phase, we run the preprocessing of Theorem 3.1 on y.

Fix a constant $c \ge 1$. If $\beta \ge 1/(8c \log n)$, then in the query phase we simply run Theorem 3.1, solving LCS exactly in time $\widetilde{O}(|x| + M) = \widetilde{O}((n + M)/\beta + 1)$.

Otherwise, denote by x' a random subsequence of x, where each letter $x[i]$ is removed independently with probability $1 - p$ (i.e., kept with probability p) for $p := 8c \log(n)/\beta$. Note that $p \le 1$ by our assumption on β. We can sample x' in expected time $O(|x'| + 1)$, since the difference from one unremoved letter to the next is geometrically distributed, and geometric random variates can be sampled in expected time $O(1)$, see, e.g., [19]. Note that this subsampling yields $\mathbb{E}[|x'|] = p|x| = \widetilde{O}(|x|/\beta)$ and $\mathbb{E}[M(x', y)] = pM = \widetilde{O}(M/\beta)$.

In the query phase, we sample x' and then run the query phase of Theorem 3.1 on x' and y. This runs in time $\widetilde{O}(|x'| + M(x', y) + 1)$, which is $\widetilde{O}((|x| + M)/\beta + 1)$ in expectation.

Finally, consider a fixed LCS of x and y, namely $z = x[i_1] \ldots x[i_L] = y[j_1] \ldots y[j_L]$ for some $i_1 < \ldots < i_L$ and $j_1 < \ldots < j_L$. Each letter $x[i_k]$ survives the subsampling to x' with probability p. Therefore, we can bound $L(x', y)$ from below by a binomial random variable $\text{Bin}(L, p)$ (the correct terminology is that $L(x', y)$ stochastically dominates $\text{Bin}(L, p)$). Since $Z = \text{Bin}(L, p)$ is a sum of independent $\{0, 1\}$-variables, multiplicative Chernoff applies and yields $\Pr[Z < \mathbb{E}[Z]/2] \le \exp(-\mathbb{E}[Z]/8)$. If $L \ge \beta$ then $\mathbb{E}[Z] = Lp \ge 2L/\beta$ and $\mathbb{E}[Z] \ge 8c \log n$, and thus $\Pr[L(x', y) \ge L/\beta] \ge 1 - n^{-c}$. Otherwise, if $L < \beta$, then we can only bound $L(x', y) \ge 0$. In both cases, we have $L(x', y) > L/\beta - 1$ with high probability. □

The above lemma behaves poorly if $L \le \beta$, due to the "-1" in the approximation guarantee. We next show that this can be avoided, at the cost of increasing the running time by an additive $\widetilde{O}(n)$.

LEMMA 4.2 (GENERALISED BASIC APPROXIMATION ALGORITHM). *Given $x, y \in \Sigma^n$ and $\beta \ge 1$, in expected time $\widetilde{O}(n + M/\beta)$ we can compute a value $\widetilde{L} \le L$ that w.h.p. satisfies $\widetilde{L} \ge L/\beta$.*

PROOF. We run the basic approximation algorithm from Lemma 4.1, which computes a value $\widetilde{L} \leq L$. Additionally, we compute the number of matching pairs $M = M(x, y)$ in time $\widetilde{O}(n)$. If $M > 0$, then there exists a matching pair, which yields a common subsequence of length 1. Therefore, if $M > 0$ we set $\widetilde{L} := \max\{\widetilde{L}, 1\}$.

In the proof of Lemma 4.1 we showed that if $L \geq \beta$ then w.h.p. we have $\widetilde{L} \geq L/\beta$. We now argue differently in the case $L < \beta$. If $L = 0$, then $\widetilde{L} \geq 0 = L/\beta$ and we are done. If $0 < L < \beta$, then there must exist at least one matching pair, so $M > 0$, so the second part of our algorithm yields $\widetilde{L} \geq 1 > L/\beta$. Hence, in all cases w.h.p. we have $\widetilde{L} \geq L/\beta$. □

We now turn towards the problem of deciding for given x, y and ℓ whether $L(x, y) \geq \ell$. To this end, we repeatedly call the basic approximation algorithm with geometrically decreasing approximation ratio β. Note that with decreasing approximation ratio we get a better approximation guarantee at the cost of higher running time. The idea is that if the LCS $L = L(x, y)$ is much shorter than the threshold ℓ, then already approximation ratio $\beta \approx \ell/L$ allows us to detect that $L < \ell$. This yields a running time bound depending on the gap L/ℓ.

LEMMA 4.3 (BASIC DECISION ALGORITHM). *Let $x, y \in \Sigma^n$. We can preprocess y in time $\widetilde{O}(n)$. Given x, the preprocessed y, and a number $1 \leq \ell \leq n$, in expected time $\widetilde{O}((n + M)L/\ell + n/\ell)$ we can w.h.p. correctly decide whether $L \geq \ell$. Our algorithm has no false positives (and w.h.p. no false negatives).*

PROOF. In the preprocessing phase, we run the preprocessing of Lemma 4.1. In the query phase, we repeatedly call the query phase of Lemma 4.1, with geometrically decreasing values of β:

(1) Preprocessing: Run the preprocessing of Lemma 4.1 on string y.
(2) For $\beta = n, n/2, n/4, \ldots, 1$:
    (3) Run the query phase of Lemma 4.1 with parameter β to obtain an estimate $\widetilde{L}$.
    (4) If $\widetilde{L} \geq \ell$: return "$L \geq \ell$"
    (5) If $\widetilde{L} \leq \ell/\beta - 1$: return "$L < \ell$"

Let us first argue correctness. Since Lemma 4.1 computes a common subsequence of x, y, we have $\widetilde{L} \leq L$. Thus, if $\widetilde{L} \geq \ell$, we correctly infer $L \geq \ell$. Moreover, w.h.p. $\widetilde{L}$ satisfies $\widetilde{L} > L/\beta - 1$. Therefore, if $\widetilde{L} \leq \ell/\beta - 1$, we can infer $L < \ell$, and this decision is correct with high probability. Finally, in the last iteration (where $\beta = 1$), we have $\ell/\beta - 1 = \ell - 1$, and thus one of $\widetilde{L} \geq \ell$ or $\widetilde{L} \leq \ell/\beta - 1$ must hold, so the algorithm indeed returns a decision.

The expected time of the query phase of Lemma 4.1 is $\widetilde{O}((n + M)/\beta + 1)$. Since β decreases geometrically, the total expected time of our algorithm is dominated by the last call.

If $L \geq \ell$, the last call is at the latest for $\beta = 1$. This yields running time $\widetilde{O}(n+M) \leq \widetilde{O}((n+M)L/\ell)$.

If $L < \ell$, note that for any $\beta \leq \frac{\ell}{L+1}$ we have $\widetilde{L} \leq L \leq \ell/\beta - 1$, and thus we return "$L < \ell$". Because we decrease β by a factor 2 in each iteration, the last call satisfies $\beta \geq \frac{\ell}{2(L+1)}$. Hence, the expected running time is $\widetilde{O}((n + M)(L + 1)/\ell + 1)$. If $L \geq 1$ then this time bound simplifies to $\widetilde{O}((n+M)L/\ell+1)$. If $L = 0$, then also $M = 0$, and the time bound becomes $\widetilde{O}(n/\ell + 1)$. In both cases we can bound the expected running time by the claimed $\widetilde{O}((n + M)L/\ell + n/\ell)$, since $\ell \leq n$. □

4.2 Approximating the Number of Matching Pairs

Recall that for given strings x, y of length n the number of matching pairs $M = M(x, y)$ can be computed in time $O(n)$, which is linear in the input size. However, later in the paper we will split x into substrings $x_1, \ldots, x_{n/m}$ and y into substrings $y_1, \ldots, y_{n/m}$, each of length m, and we will need estimates of the numbers of matching pairs $M_{ij} = M(x_i, y_j)$. In this setting, the input size is

still n (the total length of all strings x_i and y_j) and the output size is $(n/m)^2$ (all numbers M_{ij}), but we are not aware of any algorithm computing the numbers M_{ij} in near-linear time in the input plus output size $\widetilde{O}(n + (n/m)^2)$. (In fact, one can show a conditional lower bound from Boolean matrix multiplication that rules out near-linear time for computing all M_{ij}'s unless the exponent of matrix multiplication is $\omega = 2$, see Appendix B.) Therefore, we devise an approximation algorithm for estimating the number of matching pairs.

LEMMA 4.4. *For* $x_1, \ldots, x_{n/m}, y_1, \ldots, y_{n/m} \in \Sigma^m$ *write* $M_{ij} = M(x_i, y_j)$ *and* $M = \sum_{i,j} M_{ij}$. *Given the strings* $x_1, \ldots, x_{n/m}, y_1, \ldots, y_{n/m}$ *and a number* $q > 0$, *we can compute values* $\widetilde{M}_{ij}$ *that w.h.p. satisfy* $M_{ij}/8 - q \le \widetilde{M}_{ij} \le 4M_{ij}$, *in total expected time* $\widetilde{O}(n + M/q)$.

This yields a near-linear-time constant-factor approximation of all *above-average* M_{ij}: By setting $q := \Theta(\frac{Mm^2}{n^2})$, in expected time $\widetilde{O}(n + (n/m)^2)$ we obtain a constant-factor approximation of all values M_{ij} with $M_{ij} \gg q$.

PROOF. The algorithm works as follows.

(1) *Graph Construction:* Build a three-layered graph G on vertex set $V(G) = L \cup U \cup R$, where L has a node i for every string x_i, R has a node j for every string y_j, and U has a node (σ, ℓ, r) for any $\sigma \in \Sigma$ and $0 \le \ell, r \le \log m$. Put an edge from $i \in L$ to $(\sigma, \ell, r) \in U$ iff $\#_\sigma(x_i) \in [2^\ell, 2^{\ell+1})$. Similarly, put an edge from $j \in R$ to $(\sigma, \ell, r) \in U$ iff $\#_\sigma(y_j) \in [2^r, 2^{r+1})$. For an illustration of this graph see Figure 1. Note that all frequencies and thus all edges of this graphs can be computed in total time $\widetilde{O}(n)$. For $i \in L$ and $j \in R$, we denote by $U_{ij} \subseteq U$ their common neighbors. Note that any $(\sigma, \ell, r) \in U_{ij}$ represents all matching pairs of symbol σ in x_i and y_j, and the number of these matching pairs is $\#_\sigma(x_i) \cdot \#_\sigma(y_j) \in [2^{\ell+r}, 2^{\ell+r+2})$.

(2) *Subsampling:* We sample a subset $\widetilde{U} \subseteq U$ by removing each node $(\sigma, \ell, r) \in U$ independently with probability $1 - p_{\ell,r}$, where $p_{\ell,r} := \min\{1, 2^{\ell+r+3}/q\}$.

(3) *Determine Common Neighbors:* For each $(\sigma, \ell, r) \in \widetilde{U}$ enumerate all pairs of neighbors $i \in L$ and $j \in R$. For each such 2-path, add (σ, ℓ, r) to an initially empty set $\widetilde{U}_{ij}$. This step computes the sets $\widetilde{U}_{ij} := U_{ij} \cap \widetilde{U}$ in time proportional to their total size.

(4) *Output:* Return the values $\widetilde{M}_{ij} := \sum_{(\sigma, \ell, r) \in \widetilde{U}_{ij}} 2^{\ell+r}/p_{\ell,r}$.

Correctness: To analyze this algorithm, we consider the numbers $\overline{M}_{ij} := \sum_{(\sigma, \ell, r) \in U_{ij}} 2^{\ell+r}$. Observe that we have $\overline{M}_{ij} \le M_{ij} \le 4\overline{M}_{ij}$, since each $(\sigma, \ell, r) \in U_{ij}$ corresponds to at least $2^{\ell+r}$ and at most $2^{\ell+r+2}$ matching pairs of x_i and y_j. It therefore suffices to show that $\widetilde{M}_{ij}$ is close to $\overline{M}_{ij}$. Using Bernoulli random variables $\text{Ber}(p_{\ell,r})$ to express whether (σ, ℓ, r) survives the subsampling, we write

$$\widetilde{M}_{ij} = \sum_{(\sigma, \ell, r) \in U_{ij}} \frac{2^{\ell+r}}{p_{\ell,r}} \cdot \text{Ber}(p_{\ell,r}).$$

This yields an expected value of $\mathbb{E}[\widetilde{M}_{ij}] = \overline{M}_{ij}$, so by Markov's inequality we obtain $\widetilde{M}_{ij} \le 4\overline{M}_{ij} \le 4M_{ij}$ with probability at least 3/4. Since $\widetilde{M}_{ij}$ is a linear combination of independent Bernoulli random variables, we can also easily express its variance as

$$\mathbb{V}[\widetilde{M}_{ij}] = \sum_{(\sigma, \ell, r) \in U_{ij}} \left(2^{\ell+r}/p_{\ell,r}\right)^2 \cdot p_{\ell,r}(1 - p_{\ell,r}) = \sum_{(\sigma, \ell, r) \in U_{ij}} 2^{\ell+r} \cdot 2^{\ell+r} \left(\frac{1}{p_{\ell,r}} - 1\right).$$
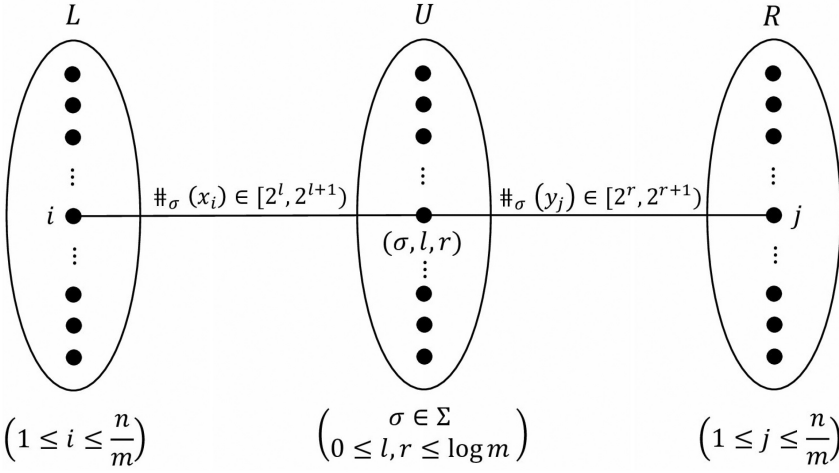
Fig. 1. Illustration of the graph G which is constructed in step 1 of the proof of Lemma 4.4. Note that if i and j are both connected to (σ, ℓ, r) (which we write as $(\sigma, \ell, r) \in U_{ij}$) then $\#_\sigma(x_i) \cdot \#_\sigma(y_j) \in [2^{\ell+r}, 2^{\ell+r+2})$.

We now use the definition of $p_{\ell,r} := \min\{1, 2^{\ell+r+3}/q\}$ to bound

$$2^{\ell+r}\left(\frac{1}{p_{\ell,r}} - 1\right) = 2^{\ell+r}\left(\max\left\{1, \frac{q}{2^{\ell+r+3}}\right\} - 1\right) = \max\{0, q/8 - 2^{\ell+r}\} \leq q/8.$$

This yields $\mathbb{V}[\widetilde{M}_{ij}] \leq \overline{M}_{ij}q/8$. We now use Chebychev's inequality $\Pr[X < \mathbb{E}[X] - \lambda] \leq \mathbb{V}[X]/\lambda^2$ on $\lambda = 0.5\mathbb{E}[X]$ and $X = \widetilde{M}_{ij}$ to obtain

$$\Pr[\widetilde{M}_{ij} < \overline{M}_{ij}/2] \leq \frac{q}{2\overline{M}_{ij}}.$$

In case $M_{ij} \geq 8q$, we have $\overline{M}_{i,j} \geq M_{ij}/4 \geq 2q$ and hence $\Pr[\widetilde{M}_{ij} \geq M_{ij}/8] \geq \Pr[\widetilde{M}_{ij} \geq \overline{M}_{ij}/2] \geq 3/4$, which follows from the above inequality. Otherwise, in case $M_{ij} < 8q$, we can only use the trivial $\widetilde{M}_{ij} \geq 0 > M_{ij}/8 - q$.

Hence, each inequality $\widetilde{M}_{ij} \leq 4M_{ij}$ and $\widetilde{M}_{ij} \geq M_{ij}/8 - q$ individually holds with probability at leat 3/4. Finally, we boost the success probability by repeating the above algorithm $O(\log n)$ times and returning for each i, j the median of all computed values $\widetilde{M}_{ij}$; this is a standard boosting technique that has a simple proof using the Chernoff bound.

Running Time: Steps 1 and 2 can be easily seen to run in time $\widetilde{O}(n)$. Steps 3 and 4 run in time proportional to the total size of all sets $\widetilde{U}_{ij}$, which we claim to be at most $8M/q$ in expectation. Over $O(\log n)$ repetitions, we obtain a total expected running time of $\widetilde{O}(n + M/q)$. (We remark that here we consider a succinct output format, where only the non-zero numbers $\widetilde{M}_{ij}$ are listed; otherwise additional time of $\widetilde{O}((n/m)^2)$ is required to output the numbers $\widetilde{M}_{ij} = 0$.)

It remains to prove the claimed bound of $\mathbb{E}[\sum_{i,j} |\widetilde{U}_{ij}|] \leq 8M/q$. Since $2^{\ell+r}/p_{\ell,r} = \max\{2^{\ell+r}, q/8\} \geq q/8$, from the definition of $\widetilde{M}_{ij} = \sum_{(\sigma,\ell,r)\in\widetilde{U}_{ij}} 2^{\ell+r}/p_{\ell,r}$ we infer $\widetilde{M}_{ij} \geq \frac{q}{8}|\widetilde{U}_{ij}|$. Therefore,

$$\mathbb{E}\left[\sum_{i,j} |\widetilde{U}_{ij}|\right] \leq \mathbb{E}\left[\frac{8}{q}\sum_{i,j}\widetilde{M}_{ij}\right] = \frac{8}{q}\sum_{i,j}\overline{M}_{ij} \leq \frac{8}{q}\sum_{i,j}M_{ij} = \frac{8M}{q}. \qquad \square$$

4.3   Single Symbol Approximation Algorithm

For strings x, y that have a large number of matchings pairs $M = M(x, y)$, some symbol must appear often in x and in y. This yields a common subsequence using (several repetitions of) a single alphabet symbol.

LEMMA 4.5 (CF. LEMMA 6.6.(II) IN [21] OR ALGORITHM 3 IN [25]). *For any $x, y \in \Sigma^n$ there exists a symbol $\sigma \in \Sigma$ that appears at least $\frac{M}{2n}$ times in x and in y. Therefore, in time $\widetilde{O}(n)$ we can compute a common subsequence of x, y of length at least $\frac{M}{2n}$. In particular, we can compute a value $\widetilde{L} \leq L$ that satisfies $\widetilde{L} \geq \frac{M}{2n}$.*

PROOF. Let k be maximal such that some symbol $\sigma \in \Sigma$ appears at least k times in x and at least k times in y. Note that for every symbol $\sigma \in \Sigma$ we have $\#_\sigma(x) \leq k$ or $\#_\sigma(y) \leq k$. We can thus bound

$$M = M(x, y) = \sum_{\sigma \in \Sigma} \#_\sigma(x) \cdot \#_\sigma(y) \leq \sum_{\sigma \in \Sigma} k \cdot \#_\sigma(y) + \sum_{\sigma \in \Sigma} \#_\sigma(x) \cdot k \leq 2kn,$$

since the frequencies $\#_\sigma(x)$ sum up to at most n, and similarly for $\#_\sigma(y)$. It follows that $k \geq \frac{M}{2n}$. Computing k, and a symbol $\sigma \in \Sigma$ attaining k, in time $\widetilde{O}(n)$ is straightforward.   □

We devise a variant of Lemma 4.5 in the following setting. For strings $x_1, \ldots, x_{n/m}$, $y_1, \ldots, y_{n/m} \in \Sigma^m$ we write $L_{ij} = L(x_i, y_j)$, $M_{ij} = M(x_i, y_j)$ and $M = \sum_{i,j} M_{ij}$. We want to find for each block (i, j) a frequent symbol in x_i and y_j, or equivalently we want to find a common subsequence of x_i and y_j using a single alphabet symbol. Similarly to Lemma 4.4, we relax Lemma 4.5 to obtain a fast running time.

LEMMA 4.6. *Given $x_1, \ldots, x_{n/m}, y_1, \ldots, y_{n/m} \in \Sigma^m$ and any $q > 0$, we can compute for each i, j a number $\widetilde{L}_{ij} \leq L_{ij}$ such that w.h.p. $\widetilde{L}_{ij} \geq \frac{M_{ij} - q}{16m}$. The algorithm runs in total expected time $\widetilde{O}(n + M/q)$.*

PROOF. We run the same algorithm as in Lemma 4.4, except that in Step 4 for each i, j with non-empty set $\widetilde{U}_{ij}$ we let $\widetilde{L}_{ij}$ be the maximum of $2^{\min\{\ell, r\}}$ over all $(\sigma, \ell, r) \in \widetilde{U}_{ij}$. For each empty set $\widetilde{U}_{ij}$, we implicitly set $\widetilde{L}_{ij} = 0$, i.e., we output a sparse representation of all non-zero values $\widetilde{L}_{ij}$.

The running time analysis is the same as in Lemma 4.4.

For the upper bound on $\widetilde{L}_{ij}$, since σ appears at least 2^ℓ times in x_i and at least 2^r times in y_j, there is a common subsequence of x_i and y_j of length at least $\widetilde{L}_{ij}$. Thus, we have $\widetilde{L}_{ij} \leq L_{ij}$.

For the lower bound on $\widetilde{L}_{ij}$, fix i, j and order the tuples $(\sigma, \ell, r) \in U_{ij}$ in non-descending order of $2^{\min\{\ell, r\}}$, obtaining an ordering $(\sigma_1, \ell_1, r_1), \ldots, (\sigma_k, \ell_k, r_k)$. For $h \in [k]$ we let

$$\mathcal{S} := \{(\sigma_1, \ell_1, r_1), \ldots, (\sigma_h, \ell_h, r_h)\} \qquad \text{and} \qquad \mathcal{L} := \{(\sigma_h, \ell_h, r_h), \ldots, (\sigma_k, \ell_k, r_k)\}.$$

Recall that $\overline{M}_{ij} = \sum_{(\sigma, \ell, r) \in U_{ij}} 2^{\ell + r}$, and observe that we can pick h with

$$\sum_{(\sigma, \ell, r) \in \mathcal{S}} 2^{\ell + r} \geq \overline{M}_{ij}/2 \qquad \text{and} \qquad \sum_{(\sigma, \ell, r) \in \mathcal{L}} 2^{\ell + r} \geq \overline{M}_{ij}/2, \tag{1}$$

here we use that $\mathcal{S}$ and $\mathcal{L}$ both contain (σ_h, ℓ_h, r_h) to obtain existence of such an h. Then we have

$$\frac{\overline{M}_{ij}}{2} \leq \sum_{(\sigma, \ell, r) \in \mathcal{S}} 2^{\ell + r} = \sum_{(\sigma, \ell, r) \in \mathcal{S}} 2^{\min\{\ell, r\}} \cdot 2^{\max\{\ell, r\}} \leq 2^{\min\{\ell_h, r_h\}} \sum_{(\sigma, \ell, r) \in \mathcal{S}} 2^{\max\{\ell, r\}}.$$

Note that for any $(\sigma, \ell, r) \in \mathcal{S}$ the symbol σ appears at least $2^{\max\{\ell, r\}}$ times in x_i or in y_j, and thus the sum on the right hand side is at most $2m$. Rearranging, this yields $2^{\min\{\ell_h, r_h\}} \geq \frac{\overline{M}_{ij}}{4m} \geq \frac{M_{ij}}{16m}$,

where we used $\overline{M}_{ij} \geq M_{ij}/4$ as in the proof of Lemma 4.4. In particular, due to our ordering we have for any $(\sigma, \ell, r) \in \mathcal{L}$:

$$2^{\min\{\ell,r\}} \geq 2^{\min\{\ell_h, r_h\}} \geq \frac{M_{ij}}{16m}. \tag{2}$$

Consider the number of nodes in $\mathcal{L}$ surviving the subsampling, i.e., $Z := |\mathcal{L} \cap \widetilde{U}_{ij}|$. If $Z > 0$, then some node in $\mathcal{L}$ survived, and thus by (2) the computed value $\widetilde{L}_{ij}$ is at least $\frac{M_{ij}}{16m}$. It thus remains to analyze $\Pr[Z > 0]$.

In case some $(\sigma, \ell, r) \in \mathcal{L}$ has $p_{\ell,r} = 1$, we have $Z > 0$ with probability 1. Otherwise all $(\sigma, \ell, r) \in \mathcal{L}$ have $p_{\ell,r} < 1$ and thus $p_{\ell,r} = 2^{\ell+r+3}/q$. In this case, we write Z as a sum of independent Bernoulli random variates in the form $Z = \sum_{(\sigma, \ell, r) \in \mathcal{L}} \mathrm{Ber}(p_{\ell,r})$. In particular,

$$\mathbb{E}[Z] = \sum_{(\sigma, \ell, r) \in \mathcal{L}} 2^{\ell+r+3}/q \overset{(1)}{\geq} \frac{4\overline{M}_{ij}}{q} \geq \frac{M_{ij}}{q}.$$

Since Z is a sum of independent $\{0, 1\}$-variables, multiplicative Chernoff applies and yields that $\Pr[Z < \mathbb{E}[Z]/2] \leq \exp(-\mathbb{E}[Z]/8)$. We thus obtain

$$\Pr[Z > 0] \geq 1 - \Pr\left[Z < \mathbb{E}[Z]/2\right] \geq 1 - \exp\left(-\mathbb{E}[Z]/8\right) \geq 1 - \exp\left(-\frac{M_{ij}}{8q}\right).$$

In case $M_{ij} \geq q$, we obtain $\Pr[Z > 0] \geq 1 - \exp(-1/8) \geq 0.1$, and thus we have $\widetilde{L}_{ij} \geq \frac{M_{ij}}{16m}$ with probability at least 0.1. Otherwise, in case $M_{ij} < q$, we can only use the trivial bound $\widetilde{L}_{ij} \geq 0 > \frac{M_{ij}-q}{16m}$. In any case, we have $\widetilde{L}_{ij} \geq \frac{M_{ij}-q}{16m}$ with probability at least 0.1. Similar to the proof of Lemma 4.4, we run $O(\log n)$ independent repetitions of this algorithm and return for each i, j the maximum of all computed values $\widetilde{L}_{ij}$, to boost the success probability and finish the proof. □

5 MAIN ALGORITHM

In this section we prove Theorem 1.1. First we show that Theorem 5.1 implies Theorem 1.1, and then in the remainder of this section we prove Theorem 5.1.

THEOREM 5.1 (MAIN RESULT, RELAXATION). *Given strings x, y of length n and a time budget $T \in [n, n^2]$, in expected time $\widetilde{O}(T)$ we can compute a number $\widetilde{L}$ such that $\widetilde{L} \leq L := L(x, y)$ and w.h.p. $\widetilde{L} \geq \widetilde{\Omega}(LT^{0.4}/n^{0.8})$.*

Note that the difference between Theorems 1.1 and 5.1 is that the latter allows *expected* running time and has an additional slack of logarithmic factors in the running time. Indeed, recall Theorem 1.1:

THEOREM 1.1. *There is a randomized algorithm that, given strings x, y of length $n \geq 1$ and a time budget $T \in [n, n^2]$, with high probability computes a multiplicative $\widetilde{O}(n^{0.8}/T^{0.4})$-approximation of the length of the LCS of x and y in time $O(T)$.*

PROOF OF THEOREM 1.1 ASSUMING THEOREM 5.1. In order to remove the *expected* running time, we abort the algorithm from Theorem 5.1 after $\widetilde{O}(T)$ time steps. By Markov's inequality, we can choose the hidden constants and logfactors such that the probability of aborting is at most $1/2$. We boost the success probability of this adapted algorithm by running $O(\log n)$ independent repetitions and returning the maximum over all computed values $\widetilde{L}$. This yields an $\widetilde{O}(n^{0.8}/T^{0.4})$-approximation with high probability in time $\widetilde{O}(T)$.

To remove the logfactors in the running time, as the first step in our algorithm we subsample the given strings x, y, keeping each symbol independently with probability $p = 1/\mathrm{polylog}(n)$, resulting in subsampled strings $\tilde{x}, \tilde{y}$. Since any common subsequence of $\tilde{x}, \tilde{y}$ is also a common subsequence

of x, y, the estimate $\widetilde{L}$ that we compute for $\tilde{x}, \tilde{y}$ satisfies $\widetilde{L} \leq L(\tilde{x}, \tilde{y}) \leq L(x, y)$. Moreover, if $L(x, y) \geq$ polylog(n) then by Chernoff bound with high probability we have $L(\tilde{x}, \tilde{y}) = \widetilde{\Omega}(L(x, y))$, so that an $\widetilde{O}(n^{0.8}/T^{0.4})$-approximation on $\tilde{x}, \tilde{y}$ also yields an $\widetilde{O}(n^{0.8}/T^{0.4})$-approximation on x, y. Otherwise, if $L(x, y) \leq$ polylog(n), then in order to compute a $\widetilde{O}(1)$-approximation it suffices to compute an LCS of length 1, which is just a matching pair and can be found in time $O(n)$ (assuming that the alphabet is $[O(n)]$).

This yields an algorithm that computes a value $\widetilde{L} \leq L$ such that w.h.p. $\widetilde{L} \geq \widetilde{\Omega}(LT^{0.4}/n^{0.8})$. The algorithm runs in time $O(T)$, and this running time bound holds deterministically, i.e., with probability 1. Hence, we proved Theorem 1.1. □

It remains to prove Theorem 5.1. Our algorithm is a combination of four methods that work well in different regimes of the problem, see Sections 5.1, 5.3, 5.4, and 5.5. We will combine these methods in Section 5.6.

5.1 Algorithm 1: Small L

Algorithm 1 works well if the LCS is short. It yields the following result.

Theorem 5.2 (Algorithm 1). *We can compute in expected time $\widetilde{O}(T)$ an estimate $\widetilde{L} \leq L$ that w.h.p. satisfies $\widetilde{L} \geq \min\{L, \sqrt{LT/n}\}$.*

Proof. Our Algorithm 1 works as follows.

(1) Run Lemma 4.5 on x and y.
(2) Run Lemma 4.2 on x and y with $\beta := \max\{1, \frac{M}{2T}\}$.
(3) Output the larger of the two common subsequence lengths computed in Steps 1 and 2.

Running Time: Step 1 runs in time $\widetilde{O}(n) = \widetilde{O}(T)$. Step 2 runs in expected time $\widetilde{O}(n + M/\beta)$. Since $\beta \geq \frac{M}{2T}$ we have $M/\beta \leq 2T$, so the expected running time is $\widetilde{O}(n + T) = \widetilde{O}(T)$.

Upper Bound: Steps 1 and 2 compute common subsequences, so the computed estimate $\widetilde{L}$ satisfies $\widetilde{L} \leq L$.

Approximation Guarantee: Note that Step 1 guarantees $\widetilde{L} \geq \frac{M}{2n}$ and Step 2 guarantees w.h.p. $\widetilde{L} \geq L/\beta$. If $M \leq 2T$ then $\beta = 1$ and $\widetilde{L} = L$, so we have solved the problem exactly. Otherwise we have $M > 2T$ and $\beta = \frac{M}{2T}$, so Step 2 guarantees w.h.p. $\widetilde{L} \geq 2LT/M$. By multiplying the two guarantees on $\widetilde{L}$ and taking square roots, we obtain w.h.p.

$$\widetilde{L} \geq \sqrt{\frac{M}{2n} \cdot \frac{2LT}{M}} = \sqrt{\frac{LT}{n}}.$$

It follows that w.h.p. $\widetilde{L} \geq \min\{L, \sqrt{LT/n}\}$. □

5.2 Block Sequences and Parameter Guessing

This section introduces some general notation and structure for the remaining algorithms.

Block Sequences: We split x into substrings $x_1, \ldots, x_{n/m}$ of length $m = n/\sqrt{T}$. Similarly, we split y into $y_1, \ldots, y_{n/m}$. A pair $(i, j) \in [n/m]^2$, corresponding to the substrings x_i, y_j, is called a *block*. For any block we write $M_{ij} = M(x_i, y_j)$ and $L_{ij} = L(x_i, y_j)$. Moreover, we write $(i, j) < (i', j')$ if and only if $i < i'$ and $j < j'$. A *block sequence* is a set $\mathcal{S} = \{(i_1, j_1), \ldots, (i_k, j_k)\}$ with $\mathcal{S} \subseteq [n/m]^2$ satisfying the monotonicity property $(i_1, j_1) < \ldots < (i_k, j_k)$. We define $\mu := \max_{\mathcal{S}} \sum_{(i,j) \in \mathcal{S}} M_{ij}$, where the maximum goes over all block sequences $\mathcal{S}$. In what follows, every algorithm will compute estimates $0 \leq \widetilde{L}_{ij} \leq L_{ij}$ and then choose a block sequence $\mathcal{S}$ to produce an overall estimate

$\widetilde{L} = \sum_{(i,j) \in S} \widetilde{L}_{ij}$. Note that this guarantees $\widetilde{L} \leq L$, as the sum $\sum_{(i,j) \in S} \widetilde{L}_{ij}$ corresponds to some (block-aligned) common subsequence of x and y. In order to get bounds in the other direction, we need to show that there always exists a block sequence of large LCS sum, i.e., a long "block-aligned common subsequence". This is shown by the following lemma.

LEMMA 5.3. *There exists a block sequence $\mathcal{G}$ of size $|\mathcal{G}| = \frac{L\sqrt{T}}{8n}$ such that for any $(i, j) \in \mathcal{G}$ we have $L_{ij} \geq \frac{L}{4\sqrt{T}}$ and $M_{ij} \leq \frac{8\mu n}{L\sqrt{T}}$. In particular, we have $\sum_{(i,j) \in \mathcal{G}} L_{ij} \geq \frac{L^2}{32n}$.*

Remark 2. Note that additional to guaranteeing a large LCS sum, the lemma also ensures that each selected block has a small number of matching pairs M_{ij}. We will use this property in Algorithm 4, where in order to compute (a large subset of) $\mathcal{G}$ we can focus on the blocks with small number of matching pairs $M_{ij} \leq \frac{8\mu n}{L\sqrt{T}}$. This guarantee on the number of matching pairs yields a favorable bound on the running time of our basic approximation algorithm.

Remark 3. Lemma 5.3 is analogous to [25, Lemma 8.2], but we improve the size of $\mathcal{G}$ and we have added the bound on M_{ij}, as discussed in the previous remark.

PROOF OF LEMMA 5.3. Let L_{ij}^* be the *contribution of block (i, j)* to the LCS. More precisely, fix an LCS z of x and y, and write $z = x[a_1] \dots x[a_L] = y[b_1] \dots y[b_L]$ for $(a_1, b_1) < \dots < (a_L, b_L)$. Then for any block (i, j), the number L_{ij}^* counts all indices k with $a_k \in ((i-1)m, im]$ and $b_k \in ((j-1)m, jm]$. Consider the set $\mathcal{A} := \{(i, j) \mid L_{ij}^* > 0\}$ consisting of all contributing blocks. From the monotonicity $(a_1, b_1) < \dots < (a_L, b_L)$ it follows that also the contributing blocks form a monotone sequence, in the sense that for any $(i, j), (i', j') \in \mathcal{A}$ we have $i \leq i'$ and $j \leq j'$, or $i' \leq i$ and $j' \leq j$. (However, these inequalities are not necessarily strict, so $\mathcal{A}$ is not necessarily a block sequence.) This monotonicity implies that there are $|\mathcal{A}| \leq 2n/m$ contributing blocks. Also note that $\sum_{(i,j) \in \mathcal{A}} L_{ij}^* = L$. Now consider the subset $\mathcal{B} = \{(i, j) \mid L_{ij}^* > \frac{Lm}{4n}\} \subseteq \mathcal{A}$. Note that the remaining blocks in total contribute

$$\sum_{(i,j) \in \mathcal{A} \setminus \mathcal{B}} L_{ij}^* \leq |\mathcal{A}| \cdot \frac{Lm}{4n} \leq \frac{2n}{m} \cdot \frac{Lm}{4n} = \frac{L}{2},$$

and thus $\mathcal{B}$ contributes $\sum_{(i,j) \in \mathcal{B}} L_{ij}^* \geq L/2$.

We now greedily pick a subset $\mathcal{C} \subseteq \mathcal{B}$ as follows. Pick any $(i, j) \in \mathcal{B}$, add (i, j) to $\mathcal{C}$, and then remove each $(i', j') \in \mathcal{B}$ with $i' = i$ or $j' = j$ from $\mathcal{B}$. Repeat until $\mathcal{B}$ is empty. By construction, $\mathcal{C}$ is a block sequence and for any $(i, j) \in \mathcal{C}$ we have $L_{ij} \geq \frac{Lm}{4n} = \frac{L}{4\sqrt{T}}$. We claim that $|\mathcal{C}| \geq \frac{L}{4m} = \frac{L\sqrt{T}}{4n}$. To see this, observe that all blocks $(i', j') \in \mathcal{B}$ with $i' = i$ in total contribute at most m, since they describe a subsequence of x_i, which has length m. Similarly, all blocks $(i', j') \in \mathcal{B}$ with $j' = j$ in total contribute at most m. Therefore, one step of the greedy procedure removes a contribution of at most $2m$. Since the total contribution is $\sum_{(i,j) \in \mathcal{B}} L_{ij}^* \geq L/2$, there are at least $\frac{L}{4m} = \frac{L\sqrt{T}}{4n}$ greedy steps. Finally, we consider the number of matching pairs. Since $\mathcal{C}$ is a block sequence, we have $\sum_{(i,j) \in \mathcal{C}} M_{ij} \leq \mu$. Thus, on average each $(i, j) \in \mathcal{C}$ has a number of matching pairs of at most $\mu/|\mathcal{C}| \leq \frac{4\mu n}{L\sqrt{T}}$. By Markov's inequality, at least half of the blocks $(i, j) \in \mathcal{C}$ have $M_{ij} \leq \frac{8\mu n}{L\sqrt{T}}$. We pick any $\frac{L\sqrt{T}}{8n}$ of these blocks to form the set $\mathcal{G} \subseteq \mathcal{C}$. The set $\mathcal{G}$ satisfies all claimed bounds. This finishes the proof. □

Parameter Guessing: We analyze our algorithms in terms of n (the length of the strings), T (the running time budget), L (the length of the LCS), as well as λ and μ, defined as

$$\lambda := \sum_{i,j} L_{ij} \qquad \text{and} \qquad \mu := \max_{\text{block seq. } \mathcal{S}} \sum_{(i,j) \in \mathcal{S}} M_{ij},$$

where the maximum goes over all block sequences $\mathcal{S}$. Note that λ is the total LCS length over all blocks and μ is the maximum total number of matching pairs along any block sequence.

The numbers n and T are part of the input, and we can assume to know M, since it can be computed in time $O(n)$. However, in order to set some parameters in our algorithms, it would be convenient to also know L, λ, μ up to constant factors (which seemingly is a contradiction, as our goal is to compute a polynomial-factor approximation of L).

We therefore run our algorithms $O(\log^3 n)$ times, once for each guess $\hat{L} = 2^i$, $\hat{\lambda} = 2^j$, and $\hat{\mu} = 2^k$. Then for at least one call we have $L/2 \leq \hat{L} \leq L$, $\lambda/2 \leq \hat{\lambda} \leq \lambda$, and $\mu/2 \leq \hat{\mu} \leq \mu$, that is, we know L, λ, μ up to constant factors. For this correct guess, we prove that our algorithms have the promised approximation guarantee and running time bound. For the wrong guesses, the approximation guarantee can fail, but we always ensure the upper bound $\widetilde{L} \leq L$, by ensuring that the estimate corresponds to some common subsequence of x and y. Hence, returning the maximum computed value $\widetilde{L}$ over all guesses $\hat{L}, \hat{\lambda}, \hat{\mu}$ yields the promised approximation guarantee. For this reason, in the following we assume to know estimates $\hat{L} \approx L$, $\hat{\lambda} \approx \lambda$, $\hat{\mu} \approx \mu$ up to constant factors; we will only use them to set certain parameters.

We remark that for the wrong guesses, not only the approximation guarantee but also the running time bound can fail, so we need to abort each of the $O(\log^3 n)$ calls after time $\widetilde{O}(T)$.

Diagonals: A *diagonal* is a set of the form $\mathcal{D}_d = \{(i, j) \in [n/m]^2 \mid i - j = d\}$. Each diagonal is a block sequence, so we have $\sum_{(i,j) \in \mathcal{D}_d} M_{ij} \leq \mu$. Note that there are $2n/m - 1 < 2\sqrt{T}$ (non-empty) diagonals. Moreover, we have $\sum_d \sum_{(i,j) \in \mathcal{D}_d} M_{ij} = M$. This yields the inequality

$$M < 2\mu\sqrt{T}. \tag{3}$$

5.3 Algorithm 2: Large L, Large μ

In this section we present Algorithm 2, which works well if μ is large, i.e., if some block sequence has a large total number of matching pairs. The algorithm makes use of the single symbol approximation that we designed in Lemma 4.6. This yields estimates $0 \leq \widetilde{L}_{ij} \leq L_{ij}$, over which we then perform dynamic programming to determine the maximum of $\sum_{(i,j) \in \mathcal{S}} \widetilde{L}_{ij}$ over all block sequences $\mathcal{S}$. (This is similar to [25, Algorithm 3], but we obtain concentration in a wider regime, see Remark 4 for a comparison.)

THEOREM 5.4 (ALGORITHM 2). *We can compute in expected time $\widetilde{O}(T)$ an estimate $\widetilde{L} \leq L$ that w.h.p. satisfies*

$$\widetilde{L} = \Omega\left(\frac{\mu\sqrt{T}}{n}\right).$$

PROOF. Algorithm 2 works as follows.

(1) Run Lemma 4.6 with $q := \frac{M}{4T}$ to compute values $\widetilde{L}_{ij}$.
(2) Perform dynamic programming over $[n/m]^2$ to determine the maximum $\sum_{(i,j) \in \mathcal{S}} \widetilde{L}_{ij}$ over all block sequences $\mathcal{S}$. Output this maximum value $\widetilde{L}$. More precisely:
   - Initialize $D[i, 0] = D[0, i] = 0$ for any $0 \leq i \leq n/m$.
   - For $i = 1, \ldots, n/m$ and $j = 1, \ldots, n/m$: $D[i, j] = \max\{\widetilde{L}_{ij} + D[i-1, j-1], \ D[i-1, j], \ D[i, j-1]\}$.
   - Output $D[n/m, n/m]$.

We analyze this algorithm in the following.

Upper Bound: Since Lemma 4.6 ensures $\widetilde{L}_{ij} \leq L_{ij}$, the dynamic programming step ensures $\widetilde{L} \leq L$.

Approximation Guarantee: Let $\mathcal{S}$ be a block sequence achieving $\sum_{(i,j)\in\mathcal{S}} M_{ij} = \mu$. Step 2 computes an estimate $\widetilde{L} \geq \sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij}$, and Lemma 4.6 yields w.h.p.

$$\widetilde{L} \geq \sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij} \geq \sum_{(i,j)\in\mathcal{S}} \frac{M_{ij} - q}{16m} = \frac{\mu}{16m} - |\mathcal{S}|\frac{q}{16m}.$$

By the monotonicity property of block sequences, we have $|\mathcal{S}| \leq n/m$. Using our definitions of $q = \frac{M}{4T}$ and $m = n/\sqrt{T}$ as well as inequality (3), we obtain

$$|\mathcal{S}|\frac{q}{16m} \leq \frac{qn}{16m^2} = \frac{M}{64n} \leq \frac{\mu\sqrt{T}}{32n}.$$

Plugging this into our bound for $\widetilde{L}$ yields

$$\widetilde{L} \geq \frac{\mu\sqrt{T}}{16n} - \frac{\mu\sqrt{T}}{32n} = \frac{\mu\sqrt{T}}{32n}.$$

Running Time: For Step 1 note that Lemma 4.6 runs in expected time $\widetilde{O}(n + M/q) = \widetilde{O}(T)$. Step 2 can be easily seen to run in time $O((n/m)^2) = O(T)$ by our choice of $m = n/\sqrt{T}$. This finishes the proof. □

Remark 4. Our Algorithm 2 is similar to [25, Algorithm 3], which works as follows. For each block (i, j), their algorithm selects a random symbol σ and uses the minimum of the frequencies $\#_\sigma(x_i), \#_\sigma(y_j)$ as the estimate $\widetilde{L}_{ij}$. It can be shown that this yields $\mathbb{E}[\widetilde{L}_{ij}] = M_{ij}/(2m)$, which is a similar lower bound as provided by Lemma 4.6, but only in expectation. The summation $\sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij}$ over a block sequence $\mathcal{S}$ then allows to apply concentration inequalities to obtain a w.h.p. error guarantee, assuming $\mu \gg m^2$.

However, in the regime $\mu \leq m^2$ the value μ could be dominated by a single block with $M_{ij} \approx \mu$. In this case, we cannot hope to get concentration by summing over many blocks. Thus, picking a random symbol per block does not suffice to obtain a w.h.p. error guarantee.

Since our improved approximation ratio makes it necessary to use Algorithm 2 in the regime $\mu \ll m^2$, their algorithm is not sufficient in our context. Thus, we replace sampling a single symbol by our new Lemma 4.6.

5.4 Algorithm 3: Large L, Small μ and Large λ

Our next algorithm works well if μ is small (i.e., every block sequence has a small total number of matching pairs) and λ is large (i.e., on average every block has a large LCS).

Let us start with the intuition. The idea is to *pick some diagonal $\mathcal{D}_d$ and run the basic approximation algorithm (Lemma 4.2) with approximation ratio $\beta = \max\{1, \mu/T\}$ on each block along the diagonal.* Since every diagonal is a block sequence, we have $\sum_{(i,j)\in\mathcal{D}_d} M_{ij} \leq \mu$, which bounds the running time of this algorithm by $\widetilde{O}(n + \sum_{(i,j)\in\mathcal{D}_d} M_{ij}/\beta) = \widetilde{O}(T)$. Moreover, this algorithm produces an estimate $\widetilde{L} \leq L$ that w.h.p. satisfies

$$\widetilde{L} \geq \sum_{(i,j)\in\mathcal{D}_d} L_{ij}/\beta.$$

Since $\sum_d \sum_{(i,j)\in\mathcal{D}_d} L_{ij} = \sum_{i,j} L_{ij} = \lambda$ and there are $O(n/m)$ diagonals, on average a diagonal $\mathcal{D}_d$ satisfies $\sum_{(i,j)\in\mathcal{D}_d} L_{ij} = \Omega(\lambda m/n) = \Omega(\lambda/\sqrt{T})$. If we pick an above-average diagonal, then we obtain an estimate

$$\widetilde{L} \geq \sum_{(i,j)\in\mathcal{D}_d} L_{ij}/\beta = \Omega\left(\frac{\lambda}{\sqrt{T}\beta}\right) = \Omega\left(\min\left\{\frac{\lambda}{\sqrt{T}}, \frac{\lambda\sqrt{T}}{\mu}\right\}\right).$$

If λ is large and μ is small, then this is a good estimate.

The main difficulty in translating this idea to an actual algorithm is how to pick the diagonal. A natural approach is to pick a random diagonal, as then the *expected* LCS sum of the diagonal is sufficiently large. However, in situations where the diagonal sums are highly unbalanced, so that λ is dominated by very few diagonals that have a very large LCS sum, a random diagonal is unlikely to have an above-average LCS sum. In this situation, a random diagonal works only with negligible probability.

Therefore, we need a sampling process that favors diagonals with a large LCS sum. To this end, we first "guess" a value g such that the sum λ is dominated by summands $L_{ij} = \Theta(g)$. We call blocks (i, j) with $L_{ij} = \Omega(g)$ *good*. Next, we sample a random good block (i_0, j_0); for this we simply keep sampling random i, j until we find a good block. Finally, we pick the diagonal $\mathcal{D}_d$ containing the "seed" block (i_0, j_0) and run the above algorithm on this diagonal. This sampling procedure favors diagonals with large LCS sum, because such diagonals contain more good blocks (i, j) to start from, and thus we are more likely to pick the "seed" (i_0, j_0) in a diagonal with a large LCS sum. This yields Theorem 5.5 below.

Remark 5. There is some similarity of our Algorithm 3 with [25, Algorithm 4], as both algorithms sum up estimates of LCS values over a diagonal. However, Hajiaghayi et al. use a uniformly random diagonal, while we use non-uniform sampling to obtain a guarantee that holds with high probability. Moreover, the method used to estimate L_{ij} is very different in both algorithms.

THEOREM 5.5 (ALGORITHM 3). *We can compute in expected time $\widetilde{O}(T)$ an estimate $\widetilde{L} \leq L$ that w.h.p. satisfies*

$$\widetilde{L} = \widetilde{\Omega}\left(\min\left\{ \frac{\lambda}{\sqrt{T}}, \frac{\lambda\sqrt{T}}{\mu} \right\} \right).$$

PROOF. Note that the theorem statement is trivial if $\lambda \leq \sqrt{T}$. Indeed, in time $O(n)$ we can compute $M = M(x, y)$. If $M = 0$ then $L = \lambda = 0$ and we return $\widetilde{L} = 0$. If $M \geq 1$, then we return $\widetilde{L} = 1$. This ensures $\widetilde{L} \leq L$, since any matching pair gives a common subsequence of length 1. Moreover, in case $\lambda \leq \sqrt{T}$ the returned value $\widetilde{L} = 1$ satisfies the approximation guarantee $\widetilde{L} = \Omega(\lambda/\sqrt{T})$. Therefore, we can assume

$$\lambda > \sqrt{T}. \tag{4}$$

Algorithm 3 repeats the following procedure $O(\log n)$ times to boost its success probability.

(1) Repeat the following for g being any power of two with $\max\{1, \hat{\lambda}/(4T)\} \leq g \leq m$:
  (2) *Sampling a good block:* Pick a random set of blocks $\mathcal{R} \subseteq [n/m]^2$ of size $O\left((gT/\hat{\lambda}) \log^2 n\right)$. For each block $(i, j) \in \mathcal{R}$, test whether $L_{ij} \geq g$ using our basic decision algorithm (Lemma 4.3). If no test was successful, then set $\widetilde{L}(g) = 0$ and continue with the next value of g. Otherwise, pick a random successfully tested block (i_0, j_0) and proceed to Step 3.
  (3) *Approximating along a diagonal:* Let $\mathcal{D}$ be the diagonal containing the block (i_0, j_0). For each $(i, j) \in \mathcal{D}$: Run our basic approximation algorithm (Lemma 4.2) with approximation ratio $\beta = \max\{1, \hat{\mu}/T\}$ on x_i, y_j to obtain an estimate $\widetilde{L}_{ij}$. Finally, $\widetilde{L}(g) := \sum_{(i,j)\in\mathcal{D}} \widetilde{L}_{ij}$ is the result of iteration g.
(4) Return $\widetilde{L} = \max_g \widetilde{L}(g)$.

Upper Bound: Again it is easy to see that $\widetilde{L} \leq L$, since Lemma 4.2 yields $\widetilde{L}_{ij} \leq L_{ij}$.

Approximation Guarantee: Let $\mathcal{B}_g$ be the set of all blocks (i, j) with $g \leq L_{ij} \leq 2g$.

CLAIM 1. *If $\lambda/2 \leq \hat{\lambda} \leq \lambda$ then for some power of two g with $\max\{1, \hat{\lambda}/(4T)\} \leq g \leq m$ we have*

$$g \cdot |\mathcal{B}_g| = \Omega(\lambda/\log m). \tag{5}$$

PROOF. Write G for the set of all powers of two g with $\max\{1, \hat{\lambda}/(4T)\} \leq g \leq m$. Note that blocks (i, j) with $L_{ij} \leq \frac{\lambda}{2T}$ in total contribute at most $\lambda/2$ to $\lambda = \sum_{i,i} L_{ij}$, since the total number of blocks is $(n/m)^2 = T$. Hence, the blocks with $L_{ij} > \frac{\lambda}{2T}$ contribute at least $\lambda/2$, that is,

$$\frac{\lambda}{2} \leq \sum_{\substack{i,j \\ L_{ij} > \lambda/(2T)}} L_{ij}.$$

Note that all blocks with $L_{ij} > \frac{\lambda}{2T}$ are covered by the sets $\mathcal{B}_g$ for powers of two $g \geq \max\{1, \lambda/(4T)\} \geq \max\{1, \hat{\lambda}/(4T)\}$. Moreover, the sets $\mathcal{B}_g$ are empty for $g > m$. Therefore, the blocks with $L_{ij} > \frac{\lambda}{2T}$ are covered by the sets $\mathcal{B}_g$ with $g \in G$, that is,

$$\frac{\lambda}{2} \leq \sum_{\substack{i,j \\ L_{ij} > \lambda/(2T)}} L_{ij} \leq \sum_{g \in G} \sum_{(i,j) \in \mathcal{B}_g} L_{ij} \leq \sum_{g \in G} 2g|\mathcal{B}_g|.$$

If for all g appearing in the sum on the right hand side we would have $g \cdot |\mathcal{B}_g| < \lambda/(4\log m + 4)$ then the right hand side would be less than $\lambda/2$, so we would obtain a contradiction. This proves the claim. □

In the following we focus on an iteration of Step 1 in which we pick a value of g as promised by Claim 1.

We call a block (i, j) *good* if $L_{ij} \geq g$, and *bad* otherwise. Note that any $(i, j) \in \mathcal{B}_g$ is good, but not every good block is in $\mathcal{B}_g$. In Step 2, we claim that the set $\mathcal{R}$ w.h.p. contains at least one good block, assuming that our guess $\hat{\lambda}$ is correct up to constant factors. Indeed, since the set $\mathcal{B}_g$ is a subset of the good blocks, the probability that $\Theta((gT/\lambda)\log^2 n)$ sampled blocks do not contain any good block is at most

$$\left(1 - \frac{|\mathcal{B}_g|}{(n/m)^2}\right)^{\Theta((gT/\lambda)\log^2 n)} \overset{(5)}{\leq} \left(1 - \frac{\lambda}{gT\log m}\right)^{\Theta((gT/\lambda)\log^2 n)} \leq \exp(-\Theta(\log n)),$$

which is negligible. For any bad block $(i, j) \in \mathcal{R}$ the test $L_{ij} \geq g$ is unsuccessful, as Lemma 4.3 has no false positives. For any good block $(i, j) \in \mathcal{R}$ w.h.p. the test is successful, and w.h.p. there is at least one good block in $\mathcal{R}$. It follows that w.h.p. Step 2 finds a good block (i_0, j_0) and proceeds to Step 3. Observe that (i_0, j_0) is chosen uniformly at random from all good blocks.

We call a diagonal *good* if it contains at least $\frac{|\mathcal{B}_g|m}{4n}$ good blocks, and *bad* otherwise. Since there are less than $2n/m$ non-empty diagonals, the number of good blocks contained in bad diagonals is at most $|\mathcal{B}_g|/2$, which is at most half of all good blocks. Therefore, at least half of all good blocks are contained in good diagonals. It follows that the uniformly random good block (i_0, j_0) lies in a good diagonal with probability at least $1/2$.

Hence, with probability at least $1/2 - o(1)$ the diagonal $\mathcal{D}$ considered in Step 3 is good, that is, it contains at least $\frac{|\mathcal{B}_g|m}{4n}$ blocks (i, j) with $L_{ij} \geq g$. Since the approximations $\widetilde{L}_{ij}$ computed in Step 3 w.h.p. satisfy $\widetilde{L}_{ij} \geq L_{ij}/\beta$, we obtain

$$\widetilde{L} \geq \frac{|\mathcal{B}_g|m}{4n} \cdot \frac{g}{\beta}.$$

Inequality (5) and the definitions $m = n/\sqrt{T}$ and $\beta = \max\{1, \hat{\mu}/T\}$ now yield

$$\widetilde{L} = \widetilde{\Omega}\left(\min\left\{ \frac{\lambda}{\sqrt{T}}, \frac{\lambda\sqrt{T}}{\hat{\mu}} \right\} \right).$$

If our guess $\hat{\mu} \approx \mu$ is correct up to a constant factor, then this yields the claimed approximation guarantee. Returning the maximum over $O(\log n)$ independent repetitions of this algorithm improves the success probability from $1/2 - o(1)$ to w.h.p.

Running Time: By Lemma 4.3, the test $L_{ij} \geq g$ runs in expected time $\widetilde{O}((m + M_{ij})L_{ij}/g + m/g) = \widetilde{O}(m^2 L_{ij}/g + m/g)$. Note that in expectation for random i, j we have $\mathbb{E}[L_{ij}] = \lambda/(n/m)^2 = \lambda/T$. Therefore, the expected running time of one test is bounded by $\widetilde{O}\left(\frac{m^2\lambda}{gT} + m/g \right)$. As Step 2 performs $O\left((gT/\hat{\lambda})\log^2 n\right)$ such tests, its expected running time is $\widetilde{O}(m^2 + mT/\lambda)$, assuming that our guess $\hat{\lambda} \approx \lambda$ is correct up to a constant factor. We now use $m^2 = n^2/T \leq T$ from $n \leq T$ and $\lambda \geq \sqrt{T} \geq n/\sqrt{T} = m$ from (4) and $n \leq T$, to bound the expected running time of Step 2 by $\widetilde{O}(T)$.

For Step 3, the expected running time is $\widetilde{O}(n + \sum_{(i,j)\in\mathcal{D}} M_{ij}/\beta)$. Since $\mathcal{D}$ is a block sequence, we have $\sum_{(i,j)\in\mathcal{D}} M_{ij} \leq \mu$. Using $\beta \geq \hat{\mu}/T = \Omega(\mu/T)$ (if our guess $\hat{\mu} \approx \mu$ is correct up to a constant factor) we can bound the expected time by $\widetilde{O}(n + T) = \widetilde{O}(T)$.

Over the $O(\log n)$ iterations of Step 1 and the $O(\log n)$ repetitions for boosting the success probability, the expected running time is still $\widetilde{O}(T)$.   □

5.5 Algorithm 4: Large L, Small μ, and Small λ

Our next algorithm works well if μ is small (i.e., every block sequence has a small total number of matching pairs), λ is small (i.e., on average every block has a small LCS), and L is large (i.e., there is a long LCS). The goal of this algorithm is to detect a sufficiently large random subset of the block sequence $\mathcal{G}$ from Lemma 5.3. To this end, we first sample a random set of blocks $\mathcal{R}$ containing each block $(i, j) \in [n/m]^2$ with probability p. Then, we use our basic decision algorithm to detect the blocks $(i, j) \in \mathcal{R}$ with $L_{ij} \geq \frac{\hat{L}}{4\sqrt{T}}$, and for these blocks we set $\widetilde{L}_{ij} = \frac{\hat{L}}{4\sqrt{T}}$, while for the remaining blocks we set $\widetilde{L}_{ij} = 0$. Finally, we perform dynamic programming to determine the maximum $\sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij}$ over all block sequences $\mathcal{S}$.

Observe that for each block in $\mathcal{G} \cap \mathcal{R}$ this algorithm sets $\widetilde{L}_{ij} = \frac{\hat{L}}{4\sqrt{T}}$, so it detects a random subset of $\mathcal{G}$. We thus obtain a p-fraction of the LCS guaranteed by the block sequence $\mathcal{G}$.

Note that in this algorithm we may focus on blocks with $M_{ij} = O(\frac{\mu n}{L\sqrt{T}})$, since this holds for all blocks in $\mathcal{G}$. Moreover, since λ is small, most blocks outside of $\mathcal{G}$ have small LCS L_{ij}. These bounds on L_{ij} and M_{ij} for the considered blocks allow us to bound the running time of the basic decision algorithm. We elaborate this algorithm in the following theorem.

THEOREM 5.6 (ALGORITHM 4). *We can compute in expected time $\widetilde{O}(T)$ an estimate $\widetilde{L} \leq L$ that w.h.p. satisfies*

$$\widetilde{L} = \Omega\left(\min\left\{ \frac{L^3}{n^2}, \frac{L^3 T}{\lambda n^2}, \frac{L^4 T}{\lambda \mu n^2} \right\} \right), \quad \text{assuming that} \quad \frac{L^2 T^{0.5}}{n^2}, \frac{L^2 T^{1.5}}{\lambda n^2}, \frac{L^3 T^{1.5}}{\lambda \mu n^2} = n^{\Omega(1)}.$$

PROOF. Algorithm 4 works as follows.

(1) Run Lemma 4.4 with $q := \frac{M}{T}$ to compute values $\widetilde{M}_{ij}$. Initialize $\widetilde{L}_{ij} = 0$ for all i, j.
(2) Run the preprocessing of the basic decision algorithm (Lemma 4.3) on each string y_j.

(3) Sample a set $\mathcal{R} \subseteq [n/m]^2$ by including each block (i,j) independently with probability

$$p := \min\left\{\frac{\hat{L}}{n}, \frac{\hat{L}T}{\hat{\lambda}n}, \frac{\hat{L}^2T}{\hat{\lambda}\hat{\mu}n}\right\}.$$

(4) For each $(i,j) \in \mathcal{R}$ with $\widetilde{M}_{ij} \leq 64\hat{\mu}n/(\hat{L}\sqrt{T})$: Run the query of the basic decision algorithm (Lemma 4.3) to test whether $L_{ij} \geq \frac{\hat{L}}{4\sqrt{T}}$. If this test is successful then set $\widetilde{L}_{ij} := \frac{\hat{L}}{4\sqrt{T}}$.

(5) Perform dynamic programming over $[n/m]^2$ to determine the maximum $\sum_{(i,j)\in\mathcal{S}} \widetilde{L}_{ij}$ over all block sequences $\mathcal{S}$. Output this maximum value $\widetilde{L}$.

Upper Bound: Since Lemma 4.3 has no false positives, we ensure $\widetilde{L}_{ij} \leq L_{ij}$ and thus $\widetilde{L} \leq L$.

Approximation Guarantee: The values $\widetilde{M}_{ij}$ computed in Step 1 w.h.p. satisfy $M_{ij}/8 - q \leq \widetilde{M}_{ij} \leq 4M_{ij}$. For all blocks $(i,j) \in \mathcal{G}$ we have $M_{ij} \leq \frac{8\mu n}{L\sqrt{T}}$ (by Lemma 5.3) and thus w.h.p. $\widetilde{M}_{ij} \leq \frac{32\mu n}{L\sqrt{T}}$. We may assume that our guesses $\hat{L}, \hat{\mu}$ satisfy $L/2 \leq \hat{L} \leq L$ and $\mu/2 \leq \hat{\mu} \leq \mu$; then we obtain $\widetilde{M}_{ij} \leq \frac{64\hat{\mu}n}{\hat{L}\sqrt{T}}$.

Therefore, each block in $\mathcal{G} \cap \mathcal{R}$ satisfies the property checked in Step 4, that is, for each such block we run the basic decision algorithm. Since for each $(i,j) \in \mathcal{G}$ we have $L_{ij} \geq \frac{L}{4\sqrt{T}} \geq \frac{\hat{L}}{4\sqrt{T}}$, in Step 4 for each block in $\mathcal{G} \cap \mathcal{R}$ w.h.p. we obtain an estimate $\widetilde{L}_{ij} = \frac{\hat{L}}{4\sqrt{T}}$. Since $\mathcal{G}$ is a block sequence, also $\mathcal{G} \cap \mathcal{R}$ is a block sequence, and thus the dynamic programming in Step 5 returns an estimate of

$$\widetilde{L} \geq \sum_{(i,j)\in\mathcal{G}\cap\mathcal{R}} \widetilde{L}_{ij} = |\mathcal{G} \cap \mathcal{R}| \cdot \frac{\hat{L}}{4\sqrt{T}}.$$

Note that the size $|\mathcal{G}\cap\mathcal{R}|$ is distributed as a binomial random variable $\text{Bin}(|\mathcal{G}|, p)$, with expectation $p|\mathcal{G}|$. Assuming that our guesses $\hat{L}, \hat{\lambda}, \hat{\mu}$ are correct up to constant factors, we have

$$p|\mathcal{G}| = \Omega\left(\min\left\{\frac{L}{n}, \frac{LT}{\lambda n}, \frac{L^2T}{\lambda\mu n}\right\} \cdot \frac{L\sqrt{T}}{n}\right) = \Omega\left(\min\left(\frac{L^2T^{0.5}}{n^2}, \frac{L^2T^{1.5}}{\lambda n^2}, \frac{L^3T^{1.5}}{\lambda\mu n^2}\right)\right) = n^{\Omega(1)},$$

by the assumption in the theorem statement. By Chernoff bound, we have

$$\Pr[|\mathcal{G} \cap \mathcal{R}| < p|\mathcal{G}|/2] \leq \exp(-p|\mathcal{G}|/8) = \exp(-n^{\Omega(1)}),$$

and thus w.h.p. we have $|\mathcal{G} \cap \mathcal{R}| \geq p|\mathcal{G}|/2$. Plugging this into our lower bound for $\widetilde{L}$ yields w.h.p.

$$\widetilde{L} \geq \frac{p|\mathcal{G}|\hat{L}}{8\sqrt{T}} = \Omega\left(\min\left\{\frac{L^3}{n^2}, \frac{L^3T}{\lambda n^2}, \frac{L^4T}{\lambda\mu n^2}\right\}\right),$$

assuming that our guesses $\hat{L}, \hat{\lambda}, \hat{\mu}$ are correct up to constant factors. This shows the claimed lower bound.

Running Time: The expected running time of Step 1 is $\widetilde{O}(n + M/q) = \widetilde{O}(T)$ since $q = \frac{M}{T}$. Step 2 runs in time $\widetilde{O}(\sum_j |y_j|) = \widetilde{O}(n)$. Steps 3 and 5 take time $O((n/m)^2) = O(T)$. In the remainder we show that Step 4 also runs in expected time $\widetilde{O}(T)$, assuming that our guesses $\hat{L}, \hat{\lambda}, \hat{\mu}$ are correct up to constant factors. Recall that w.h.p. $M_{ij}/8 - q \leq \widetilde{M}_{ij} \leq 4M_{ij}$; we condition on this event in the following.[6] Then $\widetilde{M}_{ij} = O(\frac{\mu n}{L\sqrt{T}})$ implies $M_{ij} = O(q + \frac{\mu n}{L\sqrt{T}})$. Using inequality (3) we have $q = \frac{M}{T} \leq \frac{2\mu}{\sqrt{T}} \leq \frac{2\mu n}{L\sqrt{T}}$, so $M_{ij} = O(\frac{\mu n}{L\sqrt{T}})$. Since only blocks (i,j) with $\widetilde{M}_{ij} = O(\frac{\mu n}{L\sqrt{T}})$ are tested,

---

[6]In the error event we bound the running time of Step 4 by $O(n^2)$. This has a negligible contribution to the expected running time of Step 4.

each invocation of the basic approximation algorithm in Step 4 runs in expected time $\widetilde{O}((|x_i| + M_{ij})L_{ij}\sqrt{T}/L + |x_i|\sqrt{T}/L) = \widetilde{O}((m + \frac{\mu n}{L\sqrt{T}})L_{ij}\sqrt{T}/L + m\sqrt{T}/L)$. Since each block is tested with probability at most p, Step 4 has an expected running time of

$$O\left(\sum_{i,j} p \cdot \left(\left(m + \frac{\mu n}{L\sqrt{T}}\right)L_{ij}\frac{\sqrt{T}}{L} + \frac{m\sqrt{T}}{L}\right)\right) = O\left(\left(\frac{n}{\sqrt{T}} + \frac{\mu n}{L\sqrt{T}}\right)\frac{p\lambda\sqrt{T}}{L} + \frac{pnT}{L}\right)$$

$$= \widetilde{O}\left(\frac{p\lambda n}{L} + \frac{p\lambda\mu n}{L^2} + \frac{pnT}{L}\right).$$

Note that our choice of $p = \Theta(\min\{\frac{L}{n}, \frac{LT}{\lambda n}, \frac{L^2 T}{\lambda\mu n}\})$ ensures that this running time is $\widetilde{O}(T)$.   □

5.6   Combining the Algorithms

Now we combine Algorithms 1-4. We show that w.h.p. at least one of these algorithms computes an estimate $\widetilde{L} = \widetilde{\Omega}(LT^{0.4}/n^{0.8})$. In other words, the combined algorithm has approximation ratio $\widetilde{O}(n^{0.8}/T^{0.4})$ with a running time budget of $\widetilde{O}(T)$ and thus prove Theorem 5.1.

THEOREM 5.1 (MAIN RESULT, RELAXATION). *Given strings x, y of length n and a time budget $T \in [n, n^2]$, in expected time $\widetilde{O}(T)$ we can compute a number $\widetilde{L}$ such that $\widetilde{L} \leq L := L(x, y)$ and w.h.p. $\widetilde{L} \geq \widetilde{\Omega}(LT^{0.4}/n^{0.8})$.*

Algorithm 1: Recall from Theorem 5.2 that Algorithm 1 w.h.p. returns $\widetilde{L} \geq \min\{L, \sqrt{LT/n}\}$. If $\widetilde{L} \geq L$ then we have solved the problem exactly, so we only need to consider the case $\widetilde{L} \geq \sqrt{LT/n} = L\sqrt{T/(Ln)}$. Assuming that $L \leq n^{0.6}T^{0.2}$, we obtain the claimed approximation guarantee $\widetilde{L} \geq LT^{0.4}/n^{0.8}$. Hence, from now on we can assume

$$L > n^{0.6}T^{0.2}. \tag{6}$$

Algorithm 2: Recall from Theorem 5.4 that Algorithm 2 w.h.p. returns $\widetilde{L} = \Omega(\mu\sqrt{T}/n)$. Assuming that $\mu \geq Ln^{0.2}/T^{0.1}$, we obtain the claimed approximation guarantee $\widetilde{L} = \Omega(LT^{0.4}/n^{0.8})$. Hence, from now on we can assume

$$\mu < \frac{Ln^{0.2}}{T^{0.1}}. \tag{7}$$

Algorithm 3: Recall from Theorem 5.5 that Algorithm 3 w.h.p. returns $\widetilde{L} = \widetilde{\Omega}(\min\{\frac{\lambda}{\sqrt{T}}, \frac{\lambda\sqrt{T}}{\mu}\})$. Assuming that $\lambda \geq L^2 T^{0.7}/n^{1.4}$, we have

$$\widetilde{L} = \widetilde{\Omega}\left(\min\left\{\frac{L^2 T^{0.2}}{n^{1.4}}, \frac{L^2 T^{1.2}}{\mu n^{1.4}}\right\}\right).$$

Bounding one factor L by (6) and μ by (7) yields

$$\widetilde{L} = \widetilde{\Omega}\left(\min\left\{\frac{LT^{0.4}}{n^{0.8}}, \frac{LT^{1.3}}{n^{1.6}}\right\}\right).$$

It remains to see that $T^{1.3}/n^{1.6} \geq T^{0.4}/n^{0.8}$, which is equivalent to $T^{0.9} \geq n^{0.8}$ and thus follows from $T \geq n$. Hence, Algorithm 3 satisfies the claimed approximation guarantee $\widetilde{L} = \widetilde{\Omega}(LT^{0.4}/n^{0.8})$, under our assumption on λ. We can thus from now on assume

$$\lambda < \frac{L^2 T^{0.7}}{n^{1.4}}. \tag{8}$$

Algorithm 4: We first verify that (6), (7), and (8) imply the assumptions of Theorem 5.6:

$$\frac{L^2 T^{0.5}}{n^2}, \frac{L^2 T^{1.5}}{\lambda n^2}, \frac{L^3 T^{1.5}}{\lambda \mu n^2} = n^{\Omega(1)}.$$

Indeed, we have

$$\frac{L^2 T^{0.5}}{n^2} \overset{(6)}{>} \frac{T^{0.9}}{n^{0.8}} \geq n^{0.1} = n^{\Omega(1)},$$

where we used $T \geq n$. Similarly, we have

$$\frac{L^2 T^{1.5}}{\lambda n^2} \overset{(8)}{>} \frac{T^{0.8}}{n^{0.6}} \geq n^{0.2} = n^{\Omega(1)},$$

and

$$\frac{L^3 T^{1.5}}{\lambda \mu n^2} \overset{(8)}{>} \frac{LT^{0.8}}{\mu n^{0.6}} \overset{(7)}{>} \frac{T^{0.9}}{n^{0.8}} \geq n^{0.1} = n^{\Omega(1)}.$$

As these assumptions hold, by Theorem 5.6 Algorithm 4 w.h.p. returns $\widetilde{L} = \Omega(\min\{\frac{L^3}{n^2}, \frac{L^3 T}{\lambda n^2}, \frac{L^4 T}{\lambda \mu n^2}\})$. We verify that assuming (6), (7), and (8) this yields the claimed approximation guarantee. Indeed, we have

$$\frac{L^3}{n^2} \overset{(6)}{>} \frac{LT^{0.4}}{n^{0.8}}.$$

Similarly, we have

$$\frac{L^3 T}{\lambda n^2} \overset{(8)}{>} \frac{LT^{0.3}}{n^{0.6}} \geq \frac{LT^{0.4}}{n^{0.8}},$$

since $T \leq n^2$. Finally, we have

$$\frac{L^4 T}{\lambda \mu n^2} \overset{(7)}{>} \frac{L^3 T^{1.1}}{\lambda n^{2.2}} \overset{(8)}{>} \frac{LT^{0.4}}{n^{0.8}}.$$

In all cases we obtain a lower bound of $\widetilde{L} = \widetilde{\Omega}(LT^{0.4}/n^{0.8})$.

In summary, we proved that w.h.p. at least one of the Algorithms 1-4 satisfies the desired approximation guarantee of $\widetilde{L} = \widetilde{\Omega}(LT^{0.4}/n^{0.8})$. This concludes the proof of Theorem 5.1.

APPENDICES

A HUNT AND SZYMANSKI'S LCS ALGORITHM

In this section we provide a proof sketch of Theorem 3.1.

THEOREM 3.1 (HUNT AND SZYMANSKI [28]). *We can preprocess a string y in time $\widetilde{O}(|y|)$. Given a string x and a preprocessed string y, we can compute their LCS in time $\widetilde{O}(|x| + M)$.*

PROOF. Since this algorithm is typically stated as running in time $\widetilde{O}(n+M)$, here for convenience we sketch the algorithm and show how to split it into preprocessing and query phase.

In the preprocessing phase, given string y we compute the set $\Sigma(y)$ of symbols occuring in y, and for each $\sigma \in \Sigma(y)$ we compute a sorted array A_σ containing the positions at which σ appears in y.

In the query phase, we are given a string x and a preprocessed string y. The algorithm builds a dynamic programming table T of length $|x| + 1$, maintaining the following invariant: After the i-th round, $T[k]$ stores the minimum j such that $L(x[1..i], y[1..j]) = k$ (or ∞ if not such j exists).

Initially, corresponding to round $i = 0$, the table T is computed by setting $T[0] = 0$ and $T[k] = \infty$ for any $k \in [|x|]$. Then in round i the goal is to match $x[i]$. Therefore, we iterate over all $j \in A_{x[i]}$ in decreasing order; note that this enumerates all positions j in y that match $x[i]$. For each such j,

we binary search for a value of k with $T[k-1] < j \leq T[k]$, and we set $T[k] = j$. This can be seen to maintain the invariant. In the end, the largest k with $T[k] \neq \infty$ is equal to $L(x, y)$. In pseudocode, this algorithm does the following.

(1) Preprocessing: Compute for each symbol $\sigma \in \Sigma(y)$ an array A_σ listing the positions at which σ appears in y, in sorted order.
(2) Initialization of T: $T[0] \leftarrow 0$, $T[k] \leftarrow \infty$ for any $k \in [|x|]$.
(3) For each i in $[|x|]$: For each j in $A_{x[i]}$ in decreasing order:
   (4) Find k such that $T[k-1] < j \leq T[k]$, and set $T[k] = j$.
(5) Return the largest k such that $T[k] \neq \infty$.

It is easy to see that the preprocessing can be implemented in time $O(|y| \log n)$ and the rest of the algorithm runs in time $O((|x| + M) \log n)$. □

B CONDITIONAL LOWER BOUND FOR COUNTING MATCHING PAIRS

Let $x_1, \ldots, x_{n/m}, y_1, \ldots, y_{n/m}$ be strings of length m. In Lemma 4.4 we developed an algorithm for approximating the number of matching pairs $M_{ij} = M(x_i, y_j)$ for all (i, j). Here we give evidence that approximation is necessary for this task, that is, no exact algorithm can compute all values M_{ij} in total time $\widetilde{O}(n + (n/m)^2)$.

To this end, we present a reduction from **Boolean Matrix Multiplication (BMM)**. In BMM we are given $N \times N$ matrices A, B and the task is to compute the $N \times N$ matrix $C = A \star B$ with $C_{ij} = \bigvee_{k \in [N]} A_{ik} \wedge B_{kj}$. A well-known reduction from BMM to standard matrix multiplication shows that BMM can be solved in time $O(N^\omega)$, where $\omega \leq 2.373$ is the exponent of matrix multiplication.

The following reduction shows that if we could compute all values M_{ij} in total time $\widetilde{O}(n + (n/m)^2)$, then BMM could be solved in time $\widetilde{O}(N^2)$, which would be a huge breakthrough for BMM.

THEOREM B.1. *Given Boolean $N \times N$ matrices A, B, in time $\widetilde{O}(N^2)$ we can construct strings $x_1, \ldots, x_N, y_1, \ldots, y_N$ of length N such that $M_{i,j} = M(x_i, y_j) > 0$ holds if and only if $(A \star B)_{ij} = 1$, that is, from the values M_{ij} we can read off the Boolean product of A and B.*

PROOF. We construct strings over the alphabet $\{k_c \mid k \in [N], c \in [3]\}$. We set $x_i[k] := k_1$ if $A_{ik} = 1$ and $x_i[k] := k_2$ otherwise, for any $i, k \in [N]$. We set $y_j[k] := k_1$ if $B_{kj} = 1$ and $y_j[k] := k_3$ otherwise, for any $j, k \in [N]$. Since k_2 and k_3 can never match, we observe that the number of matching pairs of x_i and y_j is equal to the number of $k \in [N]$ with $A_{ik} = B_{kj} = 1$. Therefore, $M_{ij} > 0$ holds if and only if $\bigvee_{k \in [N]} A_{ik} \wedge B_{kj}$ is true, which proves the claim. □

ACKNOWLEDGMENTS

REFERENCES

[1] Amir Abboud and Arturs Backurs. 2017. Towards hardness of approximation for polynomial time problems. In *ITCS (LIPIcs, Vol. 67)*. 11:1–11:26.
[2] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight hardness results for LCS and other sequence similarity measures. In *FOCS*. IEEE, 59–78.
[3] Amir Abboud and Karl Bringmann. 2018. Tighter connections between Formula-SAT and shaving logs. In *ICALP (LIPIcs, Vol. 107)*. 8:1–8:18.
[4] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. 2016. Simulating branching programs with edit distance and friends: Or: A polylog shaved is a lower bound made. In *STOC*. ACM, 375–388.

[5] Amir Abboud and Aviad Rubinstein. 2018. Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In *ITCS (LIPIcs, Vol. 94)*. 35:1–35:14.

[6] Alfred V. Aho, Daniel S. Hirschberg, and Jeffrey D. Ullman. 1976. Bounds on the complexity of the longest common subsequence problem. *J. ACM* 23, 1 (1976), 1–12.

[7] Shyan Akmal and Virginia Vassilevska Williams. 2021. Improved approximation for longest common subsequence over small alphabets. In *ICALP (LIPIcs, Vol. 198)*. 13:1–13:18.

[8] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. 2010. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *FOCS*. IEEE, 377–386.

[9] Alexandr Andoni and Negev Shekel Nosatzki. 2020. Edit distance in near-linear time: It's a constant factor. In *FOCS*. IEEE, 990–1001.

[10] Alexandr Andoni and Krzysztof Onak. 2012. Approximating edit distance in near-linear time. *SIAM J. Comput.* 41, 6 (2012), 1635–1648.

[11] Alberto Apostolico. 1986. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Inf. Process. Lett.* 23, 2 (1986), 63–69.

[12] Alberto Apostolico and Concettina Guerra. 1987. The longest common subsequence problem revisited. *Algorithmica* 2 (1987), 316–336.

[13] Arturs Backurs and Piotr Indyk. 2015. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *STOC*. ACM, 51–58.

[14] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. 2004. Approximating edit distance efficiently. In *FOCS*. IEEE, 550–559.

[15] Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. 2003. A sublinear algorithm for weakly approximating edit distance. In *STOC*. ACM, 316–324.

[16] Tugkan Batu, Funda Ergün, and Süleyman Cenk Sahinalp. 2006. Oblivious string embeddings and edit distance approximations. In *SODA*. ACM, 792–801.

[17] Lasse Bergroth, Harri Hakonen, and Timo Raita. 2000. A survey of longest common subsequence algorithms. In *SPIRE*. IEEE, 39–48.

[18] Joshua Brakensiek and Aviad Rubinstein. 2020. Constant-factor approximation of near-linear edit distance in near-linear time. In *STOC*. ACM, 685–698.

[19] Karl Bringmann and Tobias Friedrich. 2013. Exact and efficient generation of geometric random variates and random graphs. In *ICALP (LNCS, Vol. 7965)*. 267–278.

[20] Karl Bringmann and Marvin Künnemann. 2015. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS*. IEEE, 79–97.

[21] Karl Bringmann and Marvin Künnemann. 2018. Multivariate fine-grained complexity of longest common subsequence. In *SODA*. SIAM, 1216–1235.

[22] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. 2018. Approximating edit distance within constant factor in truly sub-quadratic time. In *FOCS*. IEEE, 979–990.

[23] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. 1992. Sparse dynamic programming I: Linear cost functions. *J. ACM* 39, 3 (1992), 519–545.

[24] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. 2019. Sublinear algorithms for gap edit distance. In *FOCS*. IEEE, 1101–1120.

[25] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. 2019. Approximating LCS in linear time: Beating the $\sqrt{n}$ barrier. In *SODA*. SIAM, 1181–1200.

[26] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. 2020. Approximating LCS in linear time: Beating the √n barrier. *CoRR* abs/2003.07285.

[27] Daniel S. Hirschberg. 1977. Algorithms for the longest common subsequence problem. *J. ACM* 24, 4 (1977), 664–675.

[28] James W. Hunt and Thomas G. Szymanski. 1977. A fast algorithm for computing longest subsequences. *Commun. ACM* 20, 5 (1977), 350–353.

[29] Costas S. Iliopoulos and Mohammad Sohel Rahman. 2009. A new efficient algorithm for computing the longest common subsequence. *Theory Comput. Syst.* 45, 2 (2009), 355–371.

[30] Michal Koucký and Michael E. Saks. 2020. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *STOC*. ACM, 699–712.

[31] William J. Masek and Mike Paterson. 1980. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.* 20, 1 (1980), 18–31.

[32] Eugene W. Myers. 1986. An $O(ND)$ difference algorithm and its variations. *Algorithmica* 1, 2 (1986), 251–266.

[33] Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. 1982. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica* 18 (1982), 171–179.

[34] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. 2019. Approximation algorithms for LCS and LIS with truly improved running times. In *FOCS*. IEEE, 1121–1145.

[35] Aviad Rubinstein and Zhao Song. 2020. Reducing approximate longest common subsequence to approximate edit distance. In *SODA*. SIAM, 1591–1600.

[36] Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *J. ACM* 21, 1 (1974), 168–173.

[37] Sun Wu, Udi Manber, Gene Myers, and Webb Miller. 1990. An $O(NP)$ sequence comparison algorithm. *Inf. Process. Lett.* 35, 6 (1990), 317–323.