Does Preprocessing Help in Fast Sequence Comparisons?

Elazar Goldenberg
elazargo@mta.ac.il
The Academic College of Tel Aviv-Yaffo
Israel

Aviad Rubinstein
Stanford University
USA
aviad@cs.stanford.edu

Barna Saha*
University of California Berkeley
USA
barnas@berkeley.edu

ABSTRACT

We study edit distance computation with preprocessing: the preprocessing algorithm acts on each string separately, and then the query algorithm takes as input the two preprocessed strings. This model is inspired by scenarios where we would like to compute edit distance between many pairs in the same pool of strings.

Our results include:

Permutation-LCS: If the LCS between two permutations has length $n - k$, we can compute it *exactly* with $O(n \log(n))$ preprocessing and $O(k \log(n))$ query time.

Small edit distance: For general strings, if their edit distance is at most k, we can compute it *exactly* with $O(n \log(n))$ preprocessing and $O(k^2 \log(n))$ query time.

Approximate edit distance: For the most general input, we can approximate the edit distance to within factor $(7 + o(1))$ with preprocessing time $\tilde{O}(n^2)$ and query time $\tilde{O}(n^{1.5+o(1)})$.

All of these results significantly improve over the state of the art in edit distance computation without preprocessing. Interestingly, by combining ideas from our algorithms with preprocessing, we provide new improved results for approximating edit distance without preprocessing in subquadratic time.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

edit distance, preprocessing, approximation algorithms

ACM Reference Format:
Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. 2020. Does Preprocessing Help in Fast Sequence Comparisons?. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357713.3384300

*Work partially supported by an NSF CAREER Award 1652303, NSF HDR TRIPODS Grant 1934846 and an Alfred P. Sloan Fellowship.

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

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384300

1 INTRODUCTION

Edit distance (aka Levenshtein distance) [37] and longest common subsequence are widely used distance measures between pairs of strings, over some alphabet Σ. They find applications in several fields like computational biology, pattern recognition, text processing, information retrieval and many more. The edit distance between A and B, denoted by $\mathrm{ED}(A, B)$, is defined as the minimum number of character insertions, deletions, and substitutions needed for converting A into B. The longest common subsequence of A and B, denoted by $\mathrm{LCS}(A, B)$, is defined as the longest subsequence common to A and B. A simple dynamic program solves this problem in quadratic time. Moreover under reasonable hardness assumptions like SETH and BP-SETH no real subquadratic time algorithm for these problems exists [2, 3, 11, 20].

While dealing with huge amounts of data (such as DNA chains, enormous storage, etc.), quadratic-time algorithms are unaffordable. This raised an active and extensive line of work on moving from quadratic-time exact computation towards (near)-linear time for approximation algorithms [6, 8, 10, 12–14, 17, 19, 22, 24, 29, 30, 44, 45], and even designing *sub*-linear time algorithms for special cases such as restriction on the distance between the input sequences [10, 13, 27] or permutations [9, 16, 25, 40, 44, 46].

In many of these applications, a large number of very long strings from a database must be compared among each other (such as comparative genomics, comparing text corpora for documents similarity etc.). For example, in *string similarity join*, which is a fundamental problem in databases, one needs to find all pairs of strings (e.g., genome sequences) in a database that are close with respect to edit distance [15].This in particular motivates developing sub-linear time algorithms. But, unfortunately even under strong assumptions, the known guarantees for sub-linear time algorithms (including recent works by the authors) are unsatisfactory. For example, recent work [27] requires $\Theta(\frac{n}{k} + k^3)$-time, and with a highly non-trivial algorithm can barely distinguish between edit distance k and k^2. Even when the strings are both permutations and k-close to each other, [9]'s nearly-optimal algorithm runs in time $\tilde{O}(\frac{n}{k} + \sqrt{n})$ and still only approximates the edit distance (to within some large constant factor). In part this is due to strong lower bounds: for example, when the edit distance is $k \ll n$, in order to have any chance of observing any difference between the strings, the algorithm must see $\Omega(\frac{n}{k})$ characters.

Our main contribution is a simple and natural augmentation to the standard model: *preprocessing*. Formally, we consider two parties that preprocess each input string independently, and then in a query phase they jointly (approximately) compute an optimal alignment. Because the preprocessing of the two strings is done independently, (i) the same preprocessing of one string can

be useful for many comparisons, and (ii) the preprocessing step can be fully parallelized in any distributed system.

In this paper we raise the question of whether preprocessing the input can accelerate the computation of the edit distance between input strings and computing their longest common subsequence. We affirmatively answer these questions by providing several algorithms that beat the state of the art algorithms where no preprocessing is allowed. Our results include faster algorithms for the tasks of exact computation of edit distance and permutation LCS. We also provide a better trade off between running time and approximation factor for edit distance approximation.

We note in particular that when the preprocessing runs in near-linear time (as is the case with all our sublinear-time algorithms), it is essentially for free in the sense that it is barely more than it took to record and store the inputs in the first place. Even when preprocessing takes super-linear time, it could be much more cost-effective to have it when dealing with large number of strings. Preprocessing captures a middle ground between (i) aforementioned works on (approximate) edit distance between two long strings; and (ii) works on approximate closest pair or nearest neighbor among a large number of short strings [4, 7, 25, 26, 31, 38, 41, 43]. Our preprocessing algorithms are most appealing when both the length and number of strings are large.

Preprocessing is also closely related to sketching [12, 15]. With an efficient sketching algorithm, we can preprocess a string to compute a small-sized sketch and then only compare the sketches during querying. The state of the art result in edit distance sketching has a preprocessing time of $\tilde{O}(nk^2)$ and query time of $poly(k \log n)$ [15]. Our algorithms get significantly better trade-offs. There are numerous works on related but different models such as computing embedding of edit distance [7, 23, 25, 41], document exchange protocols [15, 28, 33] and error-correcting codes for insertions and deletions [18, 28, 29].

1.1 Contributions

In the preprocessing model we provide much faster and simpler[1] algorithms that output much better alignments:

Permutation-LCS If the LCS between two permutations has length $n - k$, we can compute it *exactly* with $O(n \log(n))$ preprocessing and $O(k \log(n))$ query time. Contrast this result with [9] where in $\tilde{O}(\frac{n}{k} + \sqrt{n})$, the ulam distance can be approximated to within a large constant factor.

Small edit distance For general strings, if their edit distance is at most k, we can compute it *exactly* with $O(n \log(n))$ preprocessing and $O(k^2 \log(n))$ query time. Contrast this result with [27] where in $\tilde{O}(\frac{n}{k} + k^3)$ time, one can distinguish if edit distance is below k or above $\Theta(k^2)$.

Approximate edit distance For the most general input, we can approximate the edit distance to within factor $(7 + o(1))$ with preprocessing time $O(n^2 \log(n))$ and query time $O(n^{1.5+o(1)})$. Contrast this result with [6] where a $f(\epsilon)$-approximation for edit distance can be computed in time $O(n^{1.5+\epsilon})$ ($f(\epsilon)$ goes to infinity as ϵ decreases).

[1] Our first two algorithms are so simple that we could fully explain both in a single STOC talk!

What if we only preprocess one string? This setting is much harder, but we can still beat state of the art without preprocessing, namely distinguish k vs $3k^2$ with $\tilde{O}(n)$ preprocessing and $\tilde{O}(n/k + k^2)$ query time.

These strong improvements run contrary to the fine-grained complexity rule of thumb that preprocessing inputs does not help [48]. We also formalize a few conditional hardness results establishing limitations of preprocessing for fast string alignment:

Exact alignment We show that assuming (BP)-SETH, even after arbitrary polynomial-time preprocessing, computing edit distance or LCS exactly requires near-quadratic query time.

Approximate edit-distance We show that if we can α-approximate edit distance in truly-subquadratic query time with arbitrary polynomial preprocessing, then we can also $(\alpha + o(1))$-approximate it in truly-subquadratic time without preprocessing (currently not known for any $\alpha < 3$).

We remark that another related hardness result is known for the case where we only preprocess one string: Abboud and Vassilevska-Williams show that even polynomial *space* (exponential time) preprocessing doesn't help to break the near-quadratic time barrier (assuming BP-SETH/poly) [5].

Approximate edit-distance without preprocessing Interestingly, using our algorithms *with* preprocessing (for small and large edit distance regime), we give the fastest algorithm for approximating edit distance within $3 + \epsilon$ approximation *without* preprocessing. Our algorithm runs in $\tilde{O}(n^{1.6+o(1)})$ time whereas the best running time so far was $\tilde{O}(n^{1.69+o(1)})$ [6].

1.2 More Context on Our Results

Below we explain how the parameters in our results compare to existing literature without preprocessing. We note that another feature of our algorithms is that they are all relatively simple. Even our most technically involved contribution, the algorithm for general edit distance, is significantly simpler than related literature (e.g. [19, 34, 44]).

Permutation-LCS. Our $O(k \log(n))$ query time is most closely related to (and inspired by) the classic $O(n \log(n))$ for longest increasing subsequence (LIS) without preprocessing. Note that for exact computation, even after arbitrary preprocessing $\Omega(k)$ bits of communication are necessary, so our running time is tight up to the $\log(n)$ factor. Contrasting to [9], we get exact result as opposed to approximation and significantly better query time bounds for $k = O(\sqrt{n})$.

Small edit distance. Our $O(k^2 \log(n))$-time algorithm is most closely related to (and inspired by) a classic $\tilde{O}(n+k^2)$-time algorithm without preprocessing. Note that our near-n^2 SETH-lower-bound for general edit distance with preprocessing extends to k^2 SETH-lower-bound by a trivial padding argument (see also [21]). Hence our running time is near-tight assuming (BP)-SETH. Contrasting to [27], we again get exact result and better query time bound for all regimes of k, even when we allow single string preprocessing.

Table 1: Taxonomy of Algorithms Approximating Edit Distance

Authors	Time	Approximation Factor	comments
[22]	$O(n^{1.714})$	$3+\epsilon$ [2]	
[6]	$\tilde{O}(n^{1.69})$	$3+\epsilon$	
This paper	$\tilde{O}(n^{1.6+\epsilon})$	$3+\epsilon$	
[6]	$\tilde{O}(n^{1.5+\epsilon})$	$f_1(\delta)$	
This paper	$\tilde{O}(n^{1.5+\epsilon})$	$7+\epsilon$	**using $\tilde{O}(n^2)$-time preprocessing**
[19, 34]	$\tilde{O}(n^{1+\delta})$	$f_2(\delta)$	$+n^{1-f_3(\delta)}$ additive error

Approximate edit distance. This result is most closely related to (and inspired by) recent subquadratic time approximation algorithms for edit distance [6, 17, 19, 22, 34]. Here, the state of the art results include a $(3+\epsilon)$-approximation in $\tilde{O}(n^{12/7})$ time [22] and later improvement to time $\tilde{O}(n^{1.69})$ [6], $f(\epsilon)$-approximation in time $O(n^{1.5+\epsilon})$ [6], or $f'(\epsilon)$-approximation in time $O(n^{1+\epsilon})$ when the true edit distance is large [19, 34] (here f, f' are functions that go to infinity as ϵ decreases). While the improvement is not as dramatic as for sublinear algorithms, after near-quadratic preprocessing, our algorithm is clearly faster than [6, 22] ($n^{1.5}$ vs $n^{1.69}$), while obtaining much better approximation guarantees than [6, 19, 34] ($7+\epsilon$ vs $f(\epsilon)$). Interestingly, this algorithm combines ideas from aforementioned recent advances on approximate edit distance computation [17, 19, 22, 42], together with our algorithm for small edit distance computation with preprocessing. Even more surprisingly, by combining ideas from our algorithms with preprocessing, we design the fastest $3+\epsilon$ approximation algorithm for edit distance without any preprocessing.

1.3 Open Problems

We now describe a couple of exciting directions for future work

Preprocess one string: An appealing variant of our preprocessing model is when only one of the string is preprocessed. (This is motivated by a scenario where a single reference string is compared to many strings that are only used once.) For sublinear algorithms, we are able to get some improvement over state of the art, but the $\Omega(n/k)$ lower bound from communication complexity continues to hold here. With subquadratic algorithms on the other hand, our preprocessing algorithm has a natural variant that could be applied to only one string. But so far we are unable to use it to obtain significant improvement over no-preprocessing approximate edit distance algorithms.

Open Question 1. What is the complexity of approximate edit distance after preprocessing one of the strings?

Approximate edit distance in sub-linear time. A natural question is whether we can combine ideas from our exact $\tilde{O}(k^2)$-time algorithm for small edit distance together with the $O(n^{1.5+o(1)})$-time approximation algorithm for general edit distance to approximate small edit distance in truly sub-k^2 time. Alternatively, it may be possible to show unconditional lower bounds (e.g. via communication complexity) for approximate edit distance in this regime.

Open Question 2. What is the complexity of approximate edit distance with preprocessing when $k \ll n$?

Beyond string alignment? As discussed before, preprocessing is particularly appealing when it runs in near-linear time and the queries run in sub-linear time. In the context of string alignment, there is a very natural notion of preprocessing where each string is preprocessed separately. An interesting, open-ended direction is to identify other problems in sub-linear algorithms where one can define preprocessing models that are both natural and allow for significant improvements.

Open Question 3. Define preprocessing models for other problems in sub-linear algorithms that are both natural and allow for significant improvements.

2 SMALL ULAM DISTANCE

In this section, we prove Theorem 2.1 where with preprocessing we can compute ulam distance (bounded by k) exactly in time $O(k\log(n))$.

Theorem 2.1 (Permutation-LCS). *Given two permutations X, Y of $\{1,\dots,n\}$ with a common string of length at least $n-k$, we can compute their LCS exactly with $O(n\log(n))$-time preprocessing and $O(k\log(n))$-time joint processing.*

Claim 1 (Structure of close permutations). *If two permutations X, Y of $\{1,\dots,n\}$ share a common string of length at least $n-k$, then they can be partitioned into $O(k)$ contiguous blocks such that each block of Y has an identical block in Y.*

Proof. The shared common string can be partitioned into at most $k+1$ blocks that are contiguous for X, and similarly for Y. The coarsest refinement of both partitions is contiguous on both X and Y and uses at most $2k+1$ blocks. □

Algorithm description. The preprocessing algorithm (Algorithm ??) constructs $\log(n)+1$ hash tables. The ℓ-th hash table corresponds to window size 2^ℓ; we use a rolling hash function (e.g. Rabin fingerprint) to construct a hash table of all contiguous substrings of X of length 2^ℓ in time $O(n)$.

Algorithm 2 finds the partition into blocks guaranteed in Claim 1. At each iteration of the algorithm, it finds the longest contiguous substring of X, starting from XStart that has an identical contiguous substring in Y. Using the prestored hashes, this is done in time $O(\log(n))$.

Finally, given the partition into blocks, we just have to solve a heaviest increasing substring problem on the $O(k)$ blocks (with weights corresponding to block lengths). This can be done in time

$O(k\log(k))$ using a standard generalization of the classic LIS algorithm (e.g. [32]). We provide pseudocode in Algorithm 3 for completeness.

In the pseudocode below we sometimes abuse notation and think of X,Y as functions from indices to characters, and similarly, we use Y^{-1} to denote the inverse of this function (i.e. given a character it returns its index in Y.

Algorithm 1: Preprocess(X)

```
n ← length(X)
for ℓ = 0 . . . log(n) do
    H[ℓ] ← Rolling hash of X with window of length 2^ℓ
return H
```

Algorithm 2: Algorithm Compress iteratively finds maximal blocks [XStart...XEnd] in X that have a matching maximal block [YStart...YEnd] in Y. At each iteration it first exponentially increases the variable ℓ until $\ell := \lfloor \log_2(\text{XEnd} - \text{XStart}) \rfloor$; it then binary searches for the exact length of the block.

```
Algorithm: Compress(X, H_X, Y, H_Y)
n ← length(X)
XStart ← 0
XBlocks ← ∅
while XStart < n do
    YStart ← Y^{-1}(X(XStart)) ;  // XStart,YStart = respective
      starts of next block
    ℓ ← 1
    while ℓ < log(n) do
        if H_X[ℓ][XStart] ∉ H_Y[ℓ] then
            break
        ℓ ← ℓ + 1
    XEnd ← XStart + 2^ℓ
    YEnd ← YStart + 2^ℓ
    while ℓ > 0 do
        ℓ ← ℓ − 1
        if H_X[ℓ][XEnd] == H_Y[ℓ][YEnd] then
            XEnd ← XEnd + 2^ℓ
            YEnd ← YEnd + 2^ℓ
    XBlocks ← XBlocks ∪ (XStart, XEnd − XStart)
    XStart ← XEnd + 1
return XBlocks
```

3 SMALL EDIT DISTANCE

In this section, we prove our result on small edit distance, when the edit distance is bounded by k. In particular, we prove Theorem 3.1.

Theorem 3.1 (Small-EDIT). *Given two strings $A = a_1a_2..a_n$ and $B = b_1b_2..b_n$ of length n over alphabet Σ, and a bound on their edit distance, $\text{ED}(A,B) \le k$, we can compute their edit distance exactly with $O(n\log(n))$-time preprocessing and $O(k^2\log(n))$-time joint processing.*

We first recall an algorithm developed in [35, 36, 39, 47] that computes edit distance in $O(n+k^2)$ time.

Algorithm 3: Algorithm HIS maintains data structure (balanced binary search tree) Pareto, which stores the total weight and Y-index of the last character of each common substring of X and Y. The data structure is maintained sorted by Y-index, and we only keep common substrings that are *pareto-optimal* (in the sense that we want common substrings that are heavier but end on lower Y-index).

```
Algorithm: HIS(XBlocks,Y)
k ← length(XBlocks)
Pareto ← new balanced binary search tree
Pareto.insert(0, 0)
for i = 1 . . . k do
    /* Add the next block to Pareto:                    */
    newY ← Y^{-1}(Xblock[i].start)
    prevY ← Pareto.prev(newY).Y
    prevWeight ← Pareto.prev(newY).weight
    newWeight ← prevWeight + Xblock.weight
    Pareto.insert(newY,newWeight)
    /* Remove old blocks that are no longer
       pareto-optimal:                                  */
    while newWeight ≥ Pareto.next(newY).weight do
        Pareto.next(newY).delete()
return Pareto.max().weight
```

Warm-up: An $O(n+k^2)$ algorithm for Edit Distance. The well-known dynamic programming algorithm computes an $(n+1)\times(n+1)$ edit-distance matrix $D[0...n][0...n]$ where entry $D[i,j]$ is the edit distance, $\text{ED}(A^i, B^j)$ between the prefixes $A[1,i]$ and $B[1,j]$ of A and B, where $A[1,i] = a_1a_2...a_i$ and $B[1,j] = b_1b_2...b_j$. The following is well-known and easy to verify coupled with the boundary condition $D[i,0] = D[0,i] = i$ for all $i \in [0,n]$.

For all $i,j \in [0,n]$

$$D[i,j] = \min\begin{cases} D[i-1,j]+1 & \text{if } i>0;\\ D[i,j-1]+1 & \text{if } j>0;\\ D[i-1,j-1]+\mathbb{1}(a_i \neq b_j) & \text{if } i,j>0.\end{cases}$$

The computation cost for this dynamic programming is $O(n^2)$. To obtain a significant cost saving when $\text{ED}(A,B) \le k << n$, the $O(n+k^2)$ algorithm works as follows. It computes the entries of D in a greedy order, computing first the entries with value $0, 1, 2, \ldots k$ respectively. Let diagonal d of matrix D, denotes all $D[i,j]$ such that $j = i+d$. Therefore, the entries with values in $[0,k]$ are located within diagonals $[-k,k]$. Now since the entries in each diagonal of D are non-decreasing, it is enough to identify for every $d \in [-k,k]$, and for all $h \in [0,k]$, the last entry of diagonal d with value h. The rest of the entries can be inferred automatically. Hence, we are overall interested in identifying at most $(2k+1)*k$ such points. The $O(n+k^2)$ algorithm shows how building a suffix tree over a combined string $A\$B$ (where \$ is a special symbol not in Σ) helps identify each of these points in $O(1)$ time, thus achieving the desired time complexity.

Let $L^h(d) = \max\{i : D[i,i+d] = h\}$. The h-wave is defined by $L^h = \langle L^h(-k), ..., L^h(k)\rangle$. Therefore, the algorithm computes L^h for $h = 0, ..k$ in the increasing order of h until a wave e is computed such that $L^e(0) = n$ (in that case $\text{ED}(A,B) = e$), or the wave L^k

is computed in the case the algorithm is thresholded by k. Given L^{h-1}, we can compute L^h as follows.

Define

$$Equal(i,d) = \max_{q \geq i} (q \mid A[i,q] = B[i+d,q])$$

Then, $L^0(0) = Equal(0,0)$ and

$$L^h(d) = \max \begin{cases} Equal(L^{h-1}(d)+1, d) & \text{if } h-1 \geq 0; \\ Equal(L^{h-1}(d-1), d) & \text{if } d-1 \geq -k, h \geq 1; \\ Equal(L^{h-1}(d+1)+1, d) & \text{if } d+1, h+1 \leq k. \end{cases}$$

Using a suffix tree of the combined string $A\$B$, any $Equal(i,d)$ query can be answered in $O(1)$ time. Next, we show that it is possible to preprocess each A and B separately so that even then each $Equal(i,d)$ query can be implemented in $O(\log n)$ time.

Preprocessing Algorithm. The preprocessing algorithm (Algorithm 1) constructs $\log(n)+1$ hash tables just like in Section 2. The ℓ-th hash table corresponds to window size 2^ℓ; we use a rolling hash function (e.g. Rabin fingerprint) to construct a hash table of all contiguous substrings of X of length 2^ℓ in time $O(n)$. Since there are $\log n + 1$ levels, the overall preprocessing time is $O(n \log n)$. Let $H_A[\ell]$ store all the hashes for windows of length 2^ℓ of A and similarly $H_B[\ell]$ stores all the hashes for windows of length 2^ℓ of B.

Answering $Equal(i,d)$ in $O(\log n)$ time. $Equal(i,d)$ queries can be implemented by doing a simple binary search over the presorted hashes in $O(\log n)$ time. The pseudocode is given below. Suppose $Equal(i,d) = q$. The first While loop (line 5-8) identifies the smallest $\ell \geq 0$ such that $q < 2^\ell$. The next While loop does a binary search for q between $i + 2^{\ell-1}$ to $i + 2^\ell$.

Algorithm 4: EQUAL(i, d, A, H_A, B, H_B)

```
1  n ← length(A)
2  AStart ← i, BStart ← i + d
3  ℓ ← 0
4  while ℓ < log(n) do
5      if H_A[ℓ][AStart] ≠ H_B[ℓ][BStart] then
6          break
7      ℓ ← ℓ + 1
8  AStart ← i + 2^{ℓ-1}, BStart ← i + d + 2^{ℓ-1}
9  AEnd ← i + 2^ℓ − 1, BEnd ← i + d + 2^ℓ − 1
10 Mid ← (AEnd−AStart+1)/2
11 while Mid ≥ 1 do
12     if H_A[Mid][AStart] == H_B[Mid][BStart] then
13         AStart ← AStart + Mid, BStart ← BStart + Mid
14     else
15         AEnd ← AEnd − Mid − 1, BEnd ← BEnd − Mid − 1
16     Mid ← (AEnd−AStart+1)/2
17 return AEnd
```

Implementing $Equal(i,d)$ query in $O(\log n)$ time together with the correctness proof of $O(n+k^2)$ algorithm leads to Theorem 3.1.

4 PREPROCESSING A SINGLE STRING: ANSWERING GAP EDIT DISTANCE IN SUBLINEAR TIME

In this section, we design an algorithm that given two strings A and B, preprocess only one string, say B. During the query phase, the string A is provided, and a query algorithm must answer whether $\text{ED}(A,B) \leq k$ or $\text{ED}(A,B) \geq 2k^2$. We give an algorithm for this quadratic gap-edit distance problem that runs in $\tilde{O}(\frac{n}{k}+k^2)$ time. Therefore, the algorithm achieves a sublinear query time whenever $k \leq n^{1/2}$ and $k \geq$polylog n. Note that this problem was recently studied in [27] without any preprocessing. They achieve a running time bound of $\tilde{O}(\frac{n}{k}+k^3)$.

4.1 Preprocessing Algorithm

Given $Y \in \Sigma^n$, we sample each index in $[1,n]$ uniformly at random with probability $\frac{\log^2 n}{k}$. Let $S = \{i_1, i_2, ..., i_s\}$ denote the sampled indices. Create the following substrings

$$B^d = b_{i_1+d} b_{i_2+d} ... b_{i_s+d}, \forall d = -k, -k+1, .., 0, ..., k-1, k.$$

By a standard application of the Chernoff bound, we can assume with probability at least $1 - \frac{1}{n^3}$, the number of sampled indices $s = \Theta(\frac{n \log^2 n}{k})$.

The preprocessing algorithm constructs $\log(s)+1$ hash tables just like in Section 3, but for each B^d, $d \in [-k,k]$. The ℓ-th hash table corresponds to window size 2^ℓ of B^d. Since there are $\log s + 1$ levels, the overall preprocessing time is $O(k * \frac{n}{k} \log^2 n \log s) = \tilde{O}(n)$ with probability $1 - \frac{1}{n^3}$. Let $H_B^d[\ell]$ store all the hashes for windows of length 2^ℓ of B^d for $d = [-k,k]$.

4.2 Query Algorithm

Given $A \in \Sigma^n$. We create a sampled substring $A_S = a_{i_1} a_{i_2} .... a_{i_s}$. We construct $\log(s)+1$ hash tables for A_S. Again, the ℓ-th hash table corresponds to window size 2^ℓ of A_S. Since there are $\log s + 1$ levels, the overall time to compute the hashes of A_S is $O(\frac{n}{k} \log^2 n \log s) = \tilde{O}(\frac{n}{k})$ with probability $1 - \frac{1}{n^3}$. Let $H_A[\ell]$ store all the hashes for windows of length 2^ℓ of A_S

We now define an approximate $Equal(i,d)$, $Approx\text{-}Equal(i,d)$ query as follows. Let $n(i) \geq i$ be the nearest index to i present in S. Define

$$Approx\text{-}Equal(i,d) = \max_{q \geq i} (q \mid q \in S, X_S[n(i),q] = Y^d[n(i),q])$$

We now run the same algorithm from Section 3 except that we replace $Equal(i,d)$ with $Approx\text{-}Equal(i,d)$. Let us use $\hat{L}^h$ to denote the h-wave computed by using $Approx\text{-}Equal(i,d)$ for $h \in [0,k]$ and $d \in [-k,k]$. If the algorithm computes $\hat{L}^h(0) \geq n$ for $h \leq k$, the algorithm returns YES. Else, it returns NO.

Clearly, the running time of the algorithm is $\tilde{O}(\frac{n}{k}+k^2)$. We now show that the algorithm solves the quadratic gap problem.

Analysis. When comparing a symbol x_i with y_j, if they do not match, we call it a 'mismatch'. The following is an easy lemma which shows we cannot miss too many mismatches due to sampling.

LEMMA 4.1. *Given $i \in [1,n]$ and $d \in [-k,k]$, let $i' \geq i$ be the smallest index such that $x_i x_{i+1} ..., x_{i'}$ and $y_{i+d} y_{i+1+d} ... y_{i'+d}$ have*

at least $k' = \frac{k}{\log n}$ *mismatches. Let* $i \le j_1, j_2, ..., j_{k'} = i'$ *be the indices such that* $x_{j_h} \neq y_{j_h} + d$. *Define a bad event* $B(i,d)$ *to be the event that none of these* k' *mismatch indices are sampled. Then* $Prob(Bad(i,d)) \le 1 - \frac{1}{n^3}$. *Moreover, all bad events are avoided with probability at least* $1 - \frac{1}{n}$.

Proof. Since the sampling probability is $\Theta(\frac{\log^2 n}{k})$, the expected number of points sampled from $j_1, j_2, .., j_{k'}$ is $\Theta(\log n)$. Now, by the Chernoff bound, the probability that none of them are sampled can be made to be $1 - \frac{1}{n^3}$ (by choosing the constants in the sampling probability appropriately).

Then by a union bound over all $i \in [1, n]$ and $d \in [-k, k]$, with probability $\ge 1 - \frac{1}{n}$ none of the bad events $Bad(i,d)$ happen. □

Therefore, we can assume all bad events are avoided. The above lemma leads to the following direct corollary.

Corollary 4.2. *For all* $i \in S$, $\ell \in [0, \log s + 1]$ *and* $d \in [-k, k]$ *if* $H_A[\ell][i] = H_B^d[\ell][i]$ *then* $a_i a_{i+1} ..., a_{i+2^\ell}$ *and* $b_{i+d} b_{i+1+d} ... b_{i+d+2^\ell}$ *have less than* $\frac{k}{\log n}$ *mismatches.*

Proof. Take any i and d. Since $Bad(i,d)$ did not happen, if $a_i a_{i+1} ..., a_{i+2^\ell}$ and $b_{i+d} b_{i+1+d} ... b_{i+d+2^\ell}$ had at least $\frac{k}{\log n}$ mismatches, we would have $H_A[\ell][i] \neq H_B^d[\ell][i]$ □

Using the above corollary, we can now show that *Approx-Equal*(i,d) is a good approximation of *Equal*(i,d).

Lemma 4.3. *If Approx-Equal*$(i,d) = q$ *then* $a_i a_{i+1} ....a_q$ *and* $b_{i+d} b_{i+d+1} ... b_{q+d}$ *have strictly less than* $2k$ *mismatches.*

Proof. Since the sampling probability is $\frac{\log^2 n}{k}$, $(n(i) - i) \le \frac{k}{\log n}$ with high probability (we assume $k \ge$ polylog n).

Now $A[n(i), q]$ can be decomposed into at most $\log s + 1 \le \log n + 1$ intervals each of length that is a power of two. Moreover for each of these intervals the computed hashes H_A and H_B^d must match. Therefore, each of these at most $\log n + 1$ intervals can have at most $\frac{k}{\log n}$ mismatches from Corollary 4.2. Thus the total number of mismatches is strictly less than $(n(i) - i) + (\log n + 1)\frac{k}{\log n} = k + \frac{2k}{\log n} \le 2k$. □

In order to complete our analysis, we now compare the h-waves computed by the exact algorithm from Section 3 and approximate h-waves computed by using *Approx-Equal*(i,d).

Lemma 4.4 (Completeness). $\forall h \in [0,k]$ *and* $d \in [-k,k]$, $\hat{L}^h(d) \ge L^h(d)$. *Therefore, if* $\mathrm{ED}(A,B) \le k$, *then the algorithm will return YES.*

Proof. The proof follows simply by induction since *Approx-Equal*$(i,d) \ge$ *Equal*(i,d). □

Lemma 4.5 (Soundness). $\forall h \in [0,k]$ *and* $d \in [-k,k]$, $\hat{L}^h(d) \le L^{2k(h+1)}(d)$. *Therefore, if* $\mathrm{ED}(A,B) > 2k^2 + 2k$, *then the algorithm will return NO.*

The proof is again by induction, and is given in the full version.

Therefore, if $\mathrm{ED}(A,B) > 2k^2 + 2k$, then $L^{2k^2+2k}(0) < n$. Then $\hat{L}^k(0) \le L^{2k^2+2k}(0) < n$, the algorithm aborts and declares NO.

Hence, we get the following theorem.

Theorem 4.6 (Small-EDIT-Single-Preprocessing). *Given two strings* $A = a_1 a_2 .. a_n$ *and,* $B = b_1 b_2 ... b_n$ *of length* n *over alphabet* Σ, *we can answer if* $\mathrm{ED}(A,B) \le k$ *or* $\mathrm{ED}(X,Y) > 3k^2$ *with probability at least* $1 - \frac{1}{n}$ *by preprocessing only a single string in* $\tilde{O}(n)$*-time and with a query time of* $\tilde{O}(\frac{n}{k} + k^2)$.

5 LARGE EDIT DISTANCE, $7 + \epsilon$-APPROX

In this section we prove our result for the large edit distance regime. Our main result is a $7 + o(1)$ approximation for $\mathrm{ED}(A,B)$ in $n^{\frac{3}{2}+o(1)}$ query time. We are allowed to preprocess each A and B separately and spend $\sim n^2$ time in overall preprocessing.

Remark (Estimating the distance vs computing an alignment). For simplicity of presentation, we write our algorithms as merely estimating the distance. It is straightforward with standard techniques to modify them to output the alignment as well in roughly the same running time.

Organization of this section. In Subsection 5.1 we give a bird's eye overview of the main technical elements of our algorithm. Subsection 5.2 formally describes the decomposition of the strings into windows, Subsection 5.3 is a standard dynamic programming for computing an optimal window-compatible matching from pairwise distances. Our main contribution is in Subsection 5.4 which describes the algorithm for learning the close-window graph.

5.1 High Level Description of the Algorithm

The basic divide-and-conquer framework for approximate edit distance. The algorithm builds upon the recent progress on approximating edit distance in subquadratic time using divide-and-conquer algorithms [6, 17, 19, 22, 34, 44], along with our small-edit-distance algorithm from Section 3. We decompose the strings A and B into contiguous substrings called *windows*. These windows can be overlapping and have variable lengths. Up to an $(1+o(1))$-factor approximation, we can now wlog restrict our attention to matchings of A to B that are "window-compatible", i.e. they respect the partition to windows (see Lemma 5.1).

If we (approximately) knew all the pairwise distances between windows, a standard DP would find an (approximately) optimal window-compatible matching efficiently (Lemma 5.3). Computing the pairwise distances is further reduced to (approximately) learning the bipartite *close-window graph*, where a pair of A- and B-windows are neighbors if their pairwise edit distance is below an appropriate threshold τ.

The goal is now to approximately learn the close-window graph while computing as few window-window distances as possible. With this in mind, we classify the windows as either *dense* (high-degree in the close window graph), or *sparse*. We use by-now-standard separate subroutines to handle each kind of windows.

Further Details of Our Algorithm. The density of a window can be estimated by computing its edit distance to a small sample of its potential neighbors. To obtain optimal tradeoff between parameters,

we cannot afford even this small sample to classify windows as dense or sparse. Here we deviate from previous works and estimate the density on-the-fly. That is each window is assumed to be sparse by default, and only when it is selected as a special "seed" for the sparse subroutine, we estimate its degree and move it to the dense subroutine if necessary. (In fact, an originally dense window can lose many of its neighbors and become sparse by the time it is selected; this does not hurt our analysis.)

The main sparse subroutine proceeds by recursively narrowing down the set of relevant candidate neighbors. Even though sparse windows take part in multiple levels of recursion, the loss in approximation from each level of the sparse subroutine is negligible, so it continues to be negligible in aggregate. The dense subroutine incurs the main loss in approximation due to the use of triangle inequality. Fortunately, each dense window can only contributes to one level of the entire recursion and thus the overall approximation factor remains bounded.

When we compute the edit distance between pairs of windows, we do it exactly using our algorithm from Section 3. This algorithm is very efficient when the windows are close, but its running time may be as slow as quadratic in the window size when the distance is large. We remark that three recent approximate edit distance algorithms [6, 19, 34] also use the basic divide-and-conquer framework, yet manage to obtain comparable or faster running times without preprocessing. Those algorithms compute window-window distances by recursively applying an approximate edit distance algorithm; while this improves efficiency, the approximation factor explodes exponentially in the depth of the recursion.

5.2 Decomposition into Variable Sized Windows

Parameters Settings. We divide the strings A, B into *windows*, equivalently contiguous substrings. We use d and t to denote the window width and the number of windows of A respectively. Fix $d = n^{1/4}$ and $t = \frac{n}{d} = n^{3/4}$ throughout the presentation.

Let $\epsilon > 0$ be an arbitrarily small constant (or slightly sub-constant), such that we would like to obtain a $(7 + O(\epsilon))$-approximation in $O(n^{3/2+O(\epsilon)})$-time. The windows in B will vary in width. Moreover, they can be overlapping where the amount of overlap will be controlled by a parameter τ which is the relative ED threshold between a pair of windows. We will vary τ geometrically, and for each value of τ, we will compute a set of windows $\mathcal{B}^\tau$. Let t_τ denote the number of windows of $\mathcal{B}^\tau$. We will have $t_\tau = O(\frac{n}{\epsilon\tau d})$.

Choice of Windows. The choice of windows play a crucial role in our overall algorithm design. For the string A, partition A into disjoint *windows* of width d denoted by $\mathcal{A}$.

$$\mathcal{A} = \{A[1, d], A[d + 1, 2d], \ldots, A[n - d + 1, n]\}$$

We now compute the windows of B. Let us take $\tau = \{0, \frac{1}{d}, \frac{(1+\epsilon)}{d}, \frac{(1+\epsilon)^2}{d}, \ldots, 1\}$. For each value of τ, we compute a set of windows $\mathcal{B}^\tau$. Finally, we set $\mathcal{B} = \cup_\tau \mathcal{B}^\tau$ to denote all computed windows of B.

For $\tau = 0$, take $h_\tau = d$, and $l_\tau = d$. For $\tau = \frac{1}{d}$, take $h_\tau = d + 1, l_\tau = d - 1$. In general, for $\tau = \frac{(1+\epsilon)^j}{d}$, $j \geq 1$, take $h_\tau = \lfloor d + (1 + \epsilon)^{j-1} \rfloor$ and $l_\tau = \lfloor d - (1 + \epsilon)^{j-1} \rfloor$.

Set $\gamma_\tau = \max(1, \lfloor \epsilon\tau d \rfloor)$. Define

$$\mathcal{H}^\tau := \{B[1, h_\tau), B[\gamma_\tau + 1, \gamma_\tau + h_\tau), B[2\gamma_\tau + 1, 2\gamma_\tau + h_\tau], \ldots.\}$$

$$\mathcal{L}^\tau := \{B[1, l_\tau), B[\gamma_\tau + 1, \gamma_\tau + l_\tau), B[2\gamma_\tau + 1, 2\gamma_\tau + l_\tau], \ldots\}$$

Finally, $\mathcal{B}\tau = \mathcal{H}^\tau \cup \mathcal{L}^\tau$, that is $\mathcal{B}\tau$ consists of intervals of length h_τ and l_τ starting at every γ_τ grid points.

For window $a \in \mathcal{A}$ (similarly for windows in $\mathcal{B}$), let $s(a)$ denote the starting index of a (e.g., $s(A[1, d]) = 1$) and let $e(a)$ denote index the the last index of a ($e(A[1, d]) = d$). This completes the description of the windows.

Note that overall we create $t = \frac{n}{d}$ windows of A and $t_\tau = O(\frac{n}{\gamma_\tau})$.

Mapping between windows. We say that a mapping $\mu : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$ between windows is monotone if for all $a, a' \in \mathcal{A}$ such that $\mu(a), \mu(a') \neq \perp$ and $s(a) \leq s(a')$ we also have that $s(\mu(a)) \leq s(\mu(a'))$ and $e(\mu(a)) \leq e(\mu(a'))$. Setting $\mu(a) = \perp$ represents deleting a from the string. As such, we define $ED(a, \perp) = d$ for all windows a.

By abuse of notation, we let $\mu \subset \mathcal{A}$ denote the set of A-windows such that $\mu(a) \neq \perp$. For $a \in \mu$, let a.next denote the window $a' \in \mu$ immediately after a (note that next depends on the mapping μ). If a is the last window in μ, we define a.next $:= \perp$. We define a.prev in the analogous way.

For a monotone mapping μ we define its *edit distance* as:

$$\mathrm{ED}(\mu) = \sum_{a \in \mathcal{A}} \mathrm{ED}(a, \mu(a)) + \sum_{i=1}^{n} \left| \#\{a \text{ s.t. } i \in \mu(a)\} - 1 \right|$$

The first term is just sum of the edit distances between matched windows. To understand the second term, notice that for each i we expect it to appear in the image $\mu(a)$ of exactly 1 window. The second term sums the difference between the number of appearances of i and 1; it is a penalty for either overlap of windows (requiring deletions) or excessive spacing (requiring insertions).

The next lemma (proof in the full version) asserts that the cost of a minimal monotone mappings provides a good approximation for the actual edit distance between the input strings A, B.

Lemma 5.1. *Let $A, B \in \Sigma^n$, then the following holds:*

1. *For every monotone mapping $\mu : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$ we have:* $\mathrm{ED}(\mu) \geq \mathrm{ED}(A, B)$.
2. *There exists a monotone mapping $\mu : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$ satisfying:* $ED(\mu) \leq (1 + 8\epsilon)\,\mathrm{ED}(A, B)$.

Low-Skew Mapping. A monotone mapping $\mu : \mathcal{A} \to \mathcal{B}$ is said to have skew at most D if for all $a, a' \in \mathcal{A}$ we have:

$$\frac{1}{D}|s(a) - s(a')| \leq |s(\mu(a)) - s(\mu(a'))| \leq D|s(a) - s(a')|$$

We next show that any monotone mapping μ can be transformed into a low-skew mapping with $D = \frac{1}{\epsilon}$ with negligible loss (proof in the full version). Along with Lemma 5.1, this ensures there exists a near-optimal low-skew mapping, which we will exploit in our algorithm design.

Lemma 5.2. *For every monotone mapping $\mu : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$, and for every $\epsilon > 0$, there exists a monotone mapping $\mu' : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$ such that:* $\mathrm{ED}(\mu') \le (1+2\epsilon)\,\mathrm{ED}(\mu)$ *and μ' has a skew that is at most $1/\epsilon$.*

Low-skew mapping will play an impartial role in our algorithm design and analysis.

5.3 Reduction to Estimating Window Costs

Let $\mathcal{E} : \mathcal{A} \times \mathcal{B} \to \{0, \ldots, d\}$ be an estimate of the edit distance such that $\mathcal{E}(a,b) \ge \mathrm{ED}(a,b)$ for all $a \in \mathcal{A}, b \in \mathcal{B}$. Given a monotone mapping $\mu : \mathcal{A} \to \mathcal{B} \cup \{\perp\}$, we define its cost with respect to $\mathcal{E}$ as follows:

$$\mathrm{ED}_{\mathcal{E}}(\mu) = \sum_{a \in \mathcal{A}} \mathcal{E}(a, \mu(a)) + \sum_{i=1}^{n} \left| \#\{a \text{ s.t. } i \in \mu(a)\} - 1 \right|$$

We define $\mathrm{ED}(\mathcal{E})$ as the minimal cost $\mathrm{ED}_{\mathcal{E}}(\mu)$ over all monotone mappings.

Now, given such an estimation the next lemma (combining ideas from [47] and [17]) asserts that one can efficiently compute $\mathrm{ED}(\mathcal{E})$.

Instead of computing $\mathrm{ED}(\mathcal{E})$ directly, we pick a threshold Δ, and verify whether $\mathrm{ED}(\mathcal{E}) \le \Delta$. Indeed, if we can answer whether $\mathrm{ED}(\mathcal{E}) \le \Delta$, or $\mathrm{ED}(\mathcal{E}) > (7 + o(1))\Delta$ efficiently, then by increasing the threshold by an $(1+\epsilon)$ factor each time, we will be able to compute $\mathrm{ED}(\mathcal{E})$ within a $(7 + o(1))$ approximation in $\log_{1+\epsilon} n$ iterations.

Lemma 5.3 (Reduction to Estimating Window Costs). *Given an estimate $\mathcal{E}$, one can compute* $\mathrm{ED}(\mathcal{E})$ *in $O(\frac{1}{\epsilon}\frac{n^2}{d^2}\log n)$-time.*

Proof. Pick a threshold Δ. Set $\gamma = \frac{\Delta d}{n}$. If $\gamma_\tau \le \gamma$, then from $\mathcal{B}^\tau$ pick every $\lfloor \frac{\gamma}{\gamma_\tau} \rfloor$ windows so that gap between two consecutive windows in $\mathcal{B}^\tau$ is $\ge \gamma - 1$ for all τ. Therefore, the total number of windows that we consider in B is $O(\frac{n}{\gamma})$.

For window w, we let $e(w)$ denote the index of the last character of w; for a set $\mathcal{W}$, $e(\mathcal{W})$ denotes the set of last indices. For $i \in e(\mathcal{W})$, we let $i.\mathrm{prev} := \max\{i' \in e(\mathcal{W}) \cup \{0\} \wedge i' < i\}$ denote the index of the previous finish in $\mathcal{W}$ (or 0 if such an index does not exist).

We abuse notation and let $\mathrm{ED}(i,j)$ denote the minimum cost of alignment ending at i, j using estimates $\mathcal{E}$. Following [47] we use dynamic programming to fill a table of $\mathrm{ED}(i,j)$ for every pair $i, j \in e(\mathcal{A}) \times e(\mathcal{B})$ such that $|i - j| \le 10\Delta$. Notice that the number of such pairs is bounded by:

$$|\mathcal{A}||\mathcal{B}|\frac{10\Delta}{n} = O\left(\frac{n}{d} \cdot \frac{n}{\gamma} \cdot \frac{\Delta}{n}\right) = O\left(\frac{n^2}{d^2}\right). \tag{1}$$

For boundary conditions, we define $\mathrm{ED}(i,j) = \infty$ whenever $|i-j| > 10\Delta$, and $\mathrm{ED}(i,0) = \mathrm{ED}(0,i) = i$.

Consider a pair i, j such that $e(a) = i$ (notice that there may be $O_\epsilon(1)$ B-windows ending at j). The cost $\mathrm{ED}(i,j)$ of alignment ending at i, j is given by taking the minimum of:

- Cost of deleting the last A-window: $i - i.\mathrm{prev} + \mathrm{ED}(i.\mathrm{prev}, j)$;
- Cost of deleting the last several B-characters: $j - j.\mathrm{prev} + \mathrm{ED}(i.\mathrm{prev}, j)$; and
- Cost of using a last pair of windows: $\min_{e(b)=j} \left\{ \mathcal{E}(a,b) + \mathrm{ED}(s(a)-1, s(b)-1) \right\}$.

Notice that the runtime of our algorithm is dominated by the number of pairs 1, aka it is $O(\frac{n^2}{d^2})$. Now, considering all choices of Δ, we get the required running time. □

5.4 Close-Window Graphs and the Preprocessing Phase

For subset of windows C we define the graph $G_{C,\tau}$ as follows: The vertex set equals C. The pair (c, c') is connected by an edge if $\mathrm{ED}(c, c') \le \tau d$. For a substring $z \in \Sigma^d$, we denote by $\mathcal{N}^{C,\tau}(z)$ the set of all windows $c \in C$ satisfying $\mathrm{ED}(c, z) \le \tau d$. The windows in $\mathcal{N}^{C,\tau}(z)$ will also be referred to as the τ-neighbors of c in C, and $|\mathcal{N}^{C,\tau}(z)|$ as its degree in C. When it is clear from the context, we will often omit τ, and simply use the terms such as neighbors and degree of c. We will abuse notation and also use this definition for $z \notin C$.

Preprocessing Phase Algorithm. The preprocessing algorithm crucially uses the algorithm for computing *small edit distance* with preprocessing from Section 3. In the preprocessing phase, strings A and B are processed separately. Let $\tau \in \{0, \frac{(1+\epsilon)^i}{d}\}$ for $i = [0..\log_{(1+\epsilon)} d]$.

Preprocessing A. For each τ, the preprocessing algorithm computes the graph $G_{\mathcal{A},\tau}$. The number of windows of A is $\frac{n}{d}$. Hence, the preprocessing time over all τ is $O(\frac{n^2}{d^2}d^2) = O(n^2)$.

Preprocessing B. The preprocessing algorithm computes $G_{\mathcal{B}^\tau,\tau}$ for each τ. By the preprocessing algorithm of Section 3, we can process entire B in $O(n \log n)$ time so that the computation of τd-thresholded edit distance between any pair of windows can be run in $O(\tau^2 d^2 \log n)$ time.

Note that for a given τ, the gap between two consecutive windows in B is $\gamma_\tau = \max(1, \lfloor \epsilon\tau d \rfloor)$. Therefore, when $\tau = 0$, the number of windows is $O(n^2)$, but for every pair of windows, edit distance computation time is $O(1)$. For $\tau > 0$, the number of windows is $O(\frac{n}{\epsilon\tau d})$. Hence, the computation time is $O(\frac{n^2}{\epsilon^2\tau^2 d^2}\tau^2 d^2 \log n) = O(\frac{n^2}{\epsilon^2})$. Thus, over all τ, the total preprocessing time for B is $\tilde{O}_\epsilon(n^2)$.

5.5 Query Phase Algorithm

The input to the query phase algorithm is the two strings, as well as the close-window graphs computed in the preprocessing phase. The output is an estimate data structure that can answer $\mathcal{E} : \mathcal{A} \times \mathcal{B} \to \mathbb{R}^+$ queries in $O(\log(n))$ time. We implicitly initialize $\mathcal{E}(a,b) \leftarrow \infty$ for all pairs (a,b).

We consider all choices of $\tau \in \{0, \frac{(1+\epsilon)^i}{d}\}$ for $i = [0..\log_{(1+\epsilon)} d]$. For τ we run the following algorithm that attempts to discover the pairs of windows $a, b \in \mathcal{A} \times \mathcal{B}^\tau$ of edit distance at most τ. The estimate algorithm has some false negatives, and it may also have false positives whose true edit distance is up to 7τ. The estimate can be fed into the DP in Section 5.3.

Recall that $t_\tau := |\mathcal{B}^\tau|$ denote the number of windows in $\mathcal{B}^\tau$, and $t_\tau = \frac{n}{\epsilon\tau d}$.

For each value of τ, the algorithm below uses $O(t_\tau^{4/3+o(1)})$ queries to edit distance of pairs of windows of length $O(d)$ of the form is $\mathrm{ED}(a,b) < \tau d$. Using the algorithm from Theorem 3.1, each query can be answered in $\tilde{O}(d^2\tau^2)$ time. Hence the total run time is given by

$$O\left(t_\tau^{4/3+o(1)} \cdot d^2\tau^2\right) = O(n^{3/2+o(1)}).$$

Initialization: Covered windows. Initially, all windows are uncovered. Intuitively, we say that a window is covered when we have upper bounded the edit distance to its relevant neighbors in $\mathcal{B}^\tau$.

A: Intervals. Consider a partition of $[n]$ into $t_\tau^{1/3+\epsilon}$ contiguous intervals of length $n/t_\tau^{1/3+\epsilon} \le t_\tau^{2/3-\epsilon} \cdot d$. For A we define the *$\mathcal{A}$-interval* $I_\mathcal{A}$ corresponding to interval $I \subset [n]$ as the set of $\le t_\tau^{2/3-\epsilon}$ windows with indices in I. Therefore, for A-windows they are either entirely contained in the interval or don't intersect it. For B, we let I/ϵ denote a $1/\epsilon$-factor expansion of I (i.e. the interval of length $|I|/\epsilon$ centered at I)[3]. We define the $\mathcal{B}^\tau$-interval $I_{\mathcal{B}^\tau}$ to be the set of windows that intersect I/ϵ. When clear from context we sometimes just call $I_\mathcal{A}, I_{\mathcal{B}^\tau}$ intervals.

B: Sampling seeds. For each $\mathcal{A}$-interval $I_\mathcal{A}$, if less than $\log^2(n)$ windows in $I_\mathcal{A}$ remain uncovered, we simply find all of their τ-neighbors in $\mathcal{B}^\tau$ using $\tilde{O}(t_\tau)$ queries and mark them covered. Otherwise we sample $\log^2(n)$ uncovered windows from $I_\mathcal{A}$. For each sampled window a, we test whether $|\mathcal{N}^{\mathcal{B}^\tau,\tau}(a)| \gtrsim t_\tau^{1/3}$. This is done by sampling $t_\tau^{2/3}\log^2(n)$ windows $b \in \mathcal{B}^\tau$ and querying $\mathrm{ED}(a,b)$ for each. (We account for those queries later, depending on whether a is dense or sparse.)

If more than $\log^2(n)/2$ of the samples belong to $\mathcal{N}^{\mathcal{B},\tau}(a)$ then a is declared dense, otherwise it is sparse. If the window is dense we process it as described below, after which it is a covered window and no longer a good sample. We then continue to sample (in random order) other uncovered windows from the same $I_\mathcal{A}$ until: If the number of sparse windows sampled so far is smaller than $\log^2(n)$, we stop sampling whenever we see $\log^2(n)$ consecutive covered windows. Otherwise (the number of observed sparse sampled windows is at least $\log^2(n)$), we stop sampling after querying $\log^2(n)$ consecutive windows which are all either sparse or covered.

Remark. Therefore, if we keep discovering dense windows, we process it as in Step C, the window gets covered, and we keep on sampling more uncovered windows.

C: Dense windows. Suppose that a is dense; choose (arbitrarily) $b \in \mathcal{N}^{\mathcal{B}^\tau,\tau}(a)$ among those discovered during Step B while processing a. For each windows-pair $a' \in \mathcal{N}^{\mathcal{A},2\tau}(a)$ and $b' \in \mathcal{N}^{\mathcal{B}^\tau,4\tau}(b)$, whose estimate has not been computed yet, the algorithm sets $\mathcal{E}(a',b') \leftarrow 7\tau$. This is done abstractly by pointing each a' to a, b' to b and marking that $\mathrm{ED}(a,b) \le \tau$. Observe that the sets $\mathcal{N}^{\mathcal{A},2\tau}(a), \mathcal{N}^{\mathcal{B}^\tau,4\tau}(b)$ have already been computed during the preprocessing phase. We mark each window in the set $\mathcal{N}^{\mathcal{A},2\tau}(a)$ as covered.

[3]For example, if $I = [20, 30]$ then its 3-expansion is $[10, 40]$.

Approximation: Observe that every $a' \in \mathcal{N}^{\mathcal{A},2\tau}(a)$ is indeed covered in the sense that by triangle inequality, for every $b'' \in \mathcal{N}^{\mathcal{B},\tau}(a')$

$$\mathrm{ED}(b,b'') \le \mathrm{ED}(b'',a') + \mathrm{ED}(a',a) + \mathrm{ED}(a,b) \le 4\tau, \qquad (2)$$

and hence $\mathcal{N}^{\mathcal{B},\tau}(a') \subseteq \mathcal{N}^{\mathcal{B},4\tau}(b)$. Similarly, by triangle inequality for every $b' \in \mathcal{N}^{\mathcal{B},4\tau}(b)$ and $a' \in \mathcal{N}^{\mathcal{B},2\tau}(a)$, we have that

$$\mathrm{ED}(a',b') \le \mathrm{ED}(a',a) + \mathrm{ED}(a,b) + \mathrm{ED}(b,b') \le 7\tau. \qquad (3)$$

Complexity: Notice that if the $(\mathcal{B}^\tau,\tau)$-neighborhoods of two dense windows (or one dense and one sparse) a and a' intersect, then when we process one of them as dense we will cover both. Hence we only need to run the dense subroutine at most $t_\tau^{2/3}$ times. Each run requires $\tilde{O}(t_\tau^{2/3})$ queries, and hence in total over the entire τ-th iteration we only need $\tilde{O}(t_\tau^{4/3})$ queries.

D: Sparse windows. For each interval $I_\mathcal{A}$, out of the set of windows $a \in I_\mathcal{A}$ which were declared sparse, we pick at random a set $S(I_\mathcal{A})$ of size $\log^2(n)$. For every window in $S(I_\mathcal{A})$, we query its entire $(\mathcal{B}^\tau,\tau)$-neighborhood using t_τ queries. For each interval $I_\mathcal{A}$ we record the union of all $\tilde{O}(t_\tau^{1/3})$ intervals $\widehat{I_{\mathcal{B}^\tau}}$ that contain any $(\mathcal{B}^\tau,\tau)$-neighbors of any of the sparse samples $a \in S(I_\mathcal{A})$. We call these $\mathcal{B}$-windows the *relevant windows* for the windows in $I_\mathcal{A}$. We henceforth no longer look to match windows from $I_\mathcal{A}$ to irrelevant $\mathcal{B}$-windows. Note that in a low-skew mapping (for more precise statement, see Lemma 5.5), windows in $I_\mathcal{A}$ cannot be mapped to any irrelevant $\mathcal{B}$-windows under that mapping. (Hence in total across all $t_\tau^{1/3+\epsilon}$ intervals the sparse samples take $\tilde{O}(t_\tau^{4/3+\epsilon})$ queries.)

Approximation: Recall that by Lemma 5.1 and Lemma 5.2, there is a low-skew monotone mapping that approximates the optimal transformation to within $(1+\epsilon)$-factor. For any low-skew monotone mapping μ, the entire interval $I_\mathcal{A}$ is mapped to a single $\mathcal{B}^\tau$-interval $I_{\mathcal{B}^\tau}$. Suppose that $(1-\epsilon)$-fraction of the sparse windows in $I_\mathcal{A}$ are mapped to $\mathcal{B}^\tau$-windows (or $\perp$) of distance greater than τ. Then we can safely discard the τ-edges for the remaining ϵ-fraction of sparse windows with negligible loss in approximation factor. Hence in total we pay only $(1+O(\epsilon))$-factor in approximation for sparse windows. Otherwise, w.h.p. at least one of the samples has a $(\mathcal{B}^\tau,\tau)$-neighbor in $I_{\mathcal{B}^\tau}$. For more details, see Lemma 5.5.

Complexity: Each uncovered $\mathcal{A}$-window has only $\tilde{O}\left(t_\tau^{1/3} \cdot t_\tau^{2/3-\epsilon}\right) = \tilde{O}(t_\tau^{1-\epsilon})$ relevant windows.

Recursion. We recurse on Parts A-D of the algorithm, with the following modifications for the ℓ-th level of the recursion.

- We increase the number of intervals to $t_\tau^{1/3+(\ell+1)\epsilon}$, and their size decreases accordingly to $\tilde{O}(t_\tau^{2/3-(\ell+1)\epsilon})$.
- We only sample relevant windows when we estimate degrees. The degree-threshold for a window to be considered "dense" remains $t_\tau^{1/3}$. Notice that a window may be dense with respect to the entire graph, but sparse with respect to its relevant windows.
- Once we discover a dense window, we run Part C without regard to relevant/irrelevant windows. In particular the calculation of total number of queries spent on dense windows

is global for the entire τ-th iteration of the algorithm, including recursion.
- For each sparse sample, we only compute the restriction of its $(\mathcal{B}^\tau, \tau)$-neighborhood to relevant windows. Hence we only spend $\tilde{O}\left(t_\tau^{1-\ell\epsilon}\right)$ queries for each sample, or a total of $\tilde{O}\left(t_\tau^{4/3+\epsilon}\right)$ queries across all intervals.
- The relevant windows for the next level of recursion are a (strict) subset of the relevant windows in the current level.

The recursion continues until each interval has less than $\log^2(n)$ windows, after which all windows are covered.

We now summarize the approximation factor and the complexity of the query algorithm.

Lemma 5.4 (Time Complexity). *Let $A, B \in \Sigma^n$. Then running time of the query phase is bounded by: $\tilde{O}_\epsilon(n^{3/2+\epsilon})$.*

Proof. Fix τ. By the complexity analysis of Step C, the total number of queries required to cover dense windows over all the recursion steps is $\tilde{O}(t_\tau^{4/3})$.

On the ℓ-th level of recursion, the number of intervals is $t_\tau^{1/3+(\ell+1)\epsilon}$. For every interval, we pick at most $\log^2 n$ sparse windows, and query all relevant windows. The number of relevant windows is $\tilde{O}(t_\tau^{1-\ell\epsilon})$. Therefore, on the ℓ-th level of recursion, the number of queries spent on sparse windows is $t_\tau^{4/3+\epsilon}$. Since the number of levels of recursion is at most $\frac{1}{\epsilon}+1$, the total number of queries spent on sparse windows is $O(\frac{t_\tau^{4/3+\epsilon}}{\epsilon})$.

Computing τd-thresholded edit distance between pairs of windows requires time $\tilde{O}(\tau^2 d^2)$ (using our algorithm from Section 3). Therefore, the time complexity for a given τ over all dense and sparse windows is $\tilde{O}(\frac{t_\tau^{4/3+\epsilon}\tau^2 d^2}{\epsilon}) = \tilde{O}(\frac{n^{3/2+\epsilon}}{\epsilon^{7/4}})$.

Since, the number of choices of τ is $O(\frac{\log n}{\epsilon})$ and the time to run the DP from Section 5.3 is $O(\frac{1}{\epsilon}n^{3/2}\log n)$, the overall total time complexity is $\tilde{O}(\frac{n^{3/2+\epsilon}}{\epsilon^{15/4}})$. □

Lemma 5.5 (Approximation). *Let $A, B \in \Sigma^n$. Let $\mathcal{E} : \mathcal{A} \times \mathcal{B} \to \mathbb{R}^+$ be the cost function produced during the query phase. Then with probability at least $1 - \frac{1}{n}$, we have:*

$$\mathrm{ED}(\mathcal{E}) \le (7+\epsilon)\,\mathrm{ED}(A, B).$$

Proof. Note that if we can construct $\mathcal{E} : \mathcal{A} \times \mathcal{B} \to \{0, \ldots, d\}$ such that $\mathrm{ED}(a,b) \le \mathcal{E}(a,b) \le 7\,\mathrm{ED}(a,b)$, then using the DP algorithm from Section 5.3 and employing Lemma 5.1, we get a $7 + o(1)$ approximation for $\mathrm{ED}(A, B)$. Moreover, by Lemma 5.2, it is only required to compute all the edges of $\mathcal{E}$ with the above accuracy which an optimum low-skew monotone mapping μ would use. Fix such a mapping μ.

We prove that for the first level of the recursion, for each interval $I \in I_A$ it is either the case that there exists a sparse window a such that: $\mu(a) \in \mathcal{N}^{\mathcal{B}_\tau,\tau}(a)$, or that the covered dense windows provide a good approximation for the edges used by μ. Indeed, fix $I \in I_A$, the proof proceeds by case analysis.

Case 1: Suppose that there exists τ such that at least ϵ-fraction of $a \in I_{\mathcal{A}}$, are such that $\mu(a) \in \mathcal{N}^{\mathcal{B}_\tau,\tau}(a)$ and a is τ-sparse.

In this case, with high probability the algorithm will eventually pick a window $a \in I_{\mathcal{A}}$ such that $\mu(a) \in \mathcal{N}^{\mathcal{B}_\tau,\tau}(a)$ and a is τ-sparse. Consider the set $\widehat{I_{\mathcal{B}^\tau}}$ recorded by the algorithm. Since μ is a low-skew mapping, one of the intervals $I_{\mathcal{B}} \in \widehat{I_{\mathcal{B}^\tau}}$ is such that all the edges $(a, \mu(a))$ where $a \in I_{\mathcal{A}}$, are such that $\mu(a) \in I_{\mathcal{B}}$, and hence declared relevant. Therefore, in further iterations of the algorithm these edges will be assigned with the required approximation guarantee.

Case 2: Suppose that for all τ at most ϵ-fraction of $a \in I_{\mathcal{A}}$, are such that $\mu(a) \in \mathcal{N}^{\mathcal{B}_\tau,\tau}(a)$ and a is τ-sparse.

In this case we may fail to detect all the edges $(a, \mu(a))$, where $\mu(a) \in \mathcal{N}^{\mathcal{B}_\tau,\tau}(a)$ and a is τ-sparse. Nevertheless, in that case, even if we map *all* these edges to $\perp$, we only lose a $(1+\epsilon)$ factor in the edit distance. As for the rest of the windows $a \in I_{\mathcal{A}}$, we claim that with high probability for at least $1-\epsilon$ of the windows a we have: $\mathcal{E}(a, \mu(a)) \le 7\,\mathrm{ED}(a, \mu(a))$.

Indeed, observe that whenever the algorithm completes step B, then it is the case that with high probability all but at most ϵ-fraction of dense windows are already covered. If this is the case, then for each covered window $a \in I_{\mathcal{A}}$ we have: $\mathcal{E}(a, \mu(a)) \le 7\,\mathrm{ED}(a, \mu(a))$. For the rest we have no guarantee on $\mathcal{E}(a, \mu(a))$. However, even if we map *all* these edges to $\perp$, we only lose a $(1+\epsilon)$ factor in the edit distance. The claim follows. □

We therefore have the following theorem.

Theorem 5.6. *Given two strings $A, B \in \Sigma^n$, we can approximate $\mathrm{ED}(A, B)$ within $7+o(1)$ approximation with probability at least $1-\frac{1}{n}$ with a preprocessing time of $\tilde{O}_\epsilon(n^2)$ and query time of $\tilde{O}_\epsilon(n^{3/2+o(1)})$.*

6 NO PREPROCESSING: $3 + o(1)$-APPROX IN $n^{1.6+o(1)}$ TIME

In this section we introduce our $(3+o(1))$-approximation algorithm for edit distance that runs in time $n^{1.6+o(1)}$ without preprocessing. At a high level, it is similar to other recent traingle-inequality based approximation algorithms for edit distance. In particular, the previous state of the art algorithm by Andoni [6] obtains a similar result when the edit distance is large (near-linear), but we can give an overall faster algorithm using the sublinear algorithm for small edit distance with preprocessing (Section 3). The preprocessing cost is negligble when we apply it once to each window, and use the sublinear algorithm to compute the distances of many pairs.

Theorem 6.1 (Approximate edit distance without preprocessing). *Given two strings $A, B \in \Sigma^n$, we can approximate $\mathrm{ED}(A, B)$ within $3+o(1)$ approximation in $\tilde{O}_\epsilon(n^{1.6+o(1)})$ time with probability at least $1-\frac{1}{n}$.*

High level idea. The algorithm enumerates over various thresholds τ. For each value of τ, the algorithm first marks all the $\mathcal{A}$-windows as τ-uncovered. Then, it uses sampling to estimate the degree of each $\mathcal{A}$ window, and classifies them as sparse or dense. It handles sparse windows similarly to Section 5. As for the dense windows, if there are few of them, it exhaustively finds their $(\mathcal{B}, \tau)$-neighbors. Otherwise, it sparsifies the set of uncovered dense windows as follows. It enumerates over the set of $\mathcal{B}$ windows: For each such a window b it estimates its degree with respect to uncovered dense windows. If the degree is large, then it computes

$\mathcal{N}^{\mathcal{A},2\tau}(b), \mathcal{N}^{\mathcal{B},\tau}(b)$, marks that the relative distance between pairs in $\mathcal{N}^{\mathcal{A},2\tau}(b) \times \mathcal{N}^{\mathcal{B},\tau}(b)$ as upper bounded by 3τ. It then moves each uncovered dense window in $\mathcal{N}^{\mathcal{A},2\tau}(b)$ to the set of covered windows. In such a way, since we remove the neighborhood of dense $\mathcal{B}$-windows, we show that the number of uncovered dense $\mathcal{A}$-windows decreases significantly. We recurse on the sparsification phase; each iteration uses a smaller degree threshold for dense $\mathcal{B}$-windows and handles fewer remaining uncovered dense $\mathcal{A}$-windows.

Similarities to Section 5 Algorithm. Similar to Section 5 and other recent approximation algorithms, we partition the input strings into windows, and consider the close-window graph where two windows share an edge if they are close in edit distance. We handle high-degree ("dense") windows using triangle inequality, and low-degree ("sparse") by iteratively focusing on narrowing intervals.

Main technical difference compared to Section 5 Algorithm. A subtle technicality of this algorithm is that in the sparsification phase, we can remove $\mathcal{B}$-windows of high degree, and all their $\mathcal{A}$-neighbors. This suffices to ensure that the remaining $\mathcal{A}$-windows are sparse *on average*. However, the analysis of the sparse case, crucially relies on *every window* being sparse. By Markov's inequality, once we decrease the average degree of the $\mathcal{A}$-window at most $t^{-\epsilon}$-fraction of them remain overly-dense. We can thus recurse on all the $\mathcal{B}$-windows and the $t^{-\epsilon}$-fraction overly-dense $\mathcal{A}$-windows, again removing the highest-degree $\mathcal{B}$-windows. After $O(1/\epsilon)$ iterations, all the dense $\mathcal{A}$-windows have been removed.

As in Section 5, we repeat the following steps for every τ in a multiplicative-$(1+\epsilon)$-net.

Parameters and notation. Following the notation of Section 5.2, we set the base window length to $d = n^{0.2}$, and the number of $\mathcal{A}$-windows is $t = n^{0.8}$; the number of windows in $\mathcal{B}_\tau$ is $t_\tau = O_\epsilon(t/\tau)$. Our algorithm will use $t_\tau^{3/2+o(1)}$ queries, each in time $\tilde{O}(d^2\tau^2)$, as well as the DP from Lemma 5.3. Hence the total running time is given by

$$\tilde{O}\left(t_\tau^{3/2+o(1)} \cdot d^2\tau^2 + \frac{n^2}{d^2}\right) = \tilde{O}(n^{1.6+o(1)}). \tag{4}$$

Our sparsification phase (Steps A-2 and B below) works in iterations, where in each iteration we cover the edges of the form (a, b) where b is a high degree vertex. In more detail, the algorithm iteratively identifies $\mathcal{B}$-windows with high degree. At the first iteration, the degree threshold is $\deg_1 := t_\tau^{1/2}$, and it decreases by t_τ^ϵ in each subsequent iteration. I.e. at the g-th iteration it is $\deg_g := t_\tau^{1/2-(g-1)\epsilon}$.

We maintain a partition of $\mathcal{A}$ into three subsets: $\mathcal{A} = \mathcal{A}_{\text{Sparse}} \cup \mathcal{A}_{\text{Bad}} \cup \mathcal{A}_{\text{Covered}}$. Initially, $|\mathcal{A}_{\text{Bad}}| \le |\mathcal{A}| = t \le t_\tau$. In each iteration of the sparsification phase, windows from $\mathcal{A}_{\text{Bad}}$ are moved to $\mathcal{A}_{\text{Covered}}$. The upper bound on $|\mathcal{A}_{\text{Bad}}|$ decreases by t_τ^ϵ-factor in each iteration.

Step A: Estimating density of $\mathcal{A}$-windows. For each $a \in \mathcal{A}$, we sample $t_\tau^{1/2-\epsilon}\log^2(n)$ $\mathcal{B}_\tau$-windows b at random and query $\text{ED}(b, a)$. We place a in $\mathcal{A}_{\text{Bad}}$ if at least $\frac{1}{2}\log^2(n)$ of the samples are within edit distance τ. Otherwise, we place it in $\mathcal{A}_{\text{Sparse}}$ and ignore it until Step C of the algorithm.

Complexity: We spend $\tilde{O}(t_\tau^{1/2})$ queries for each $a \in \mathcal{A}_{\text{Bad}}$, hence a total of $\tilde{O}(t_\tau^{3/2})$.

Step B-g. An iteration of the sparsification phase. In each iteration of the sparsification phase, we enumerate over the $\mathcal{B}_\tau$-windows. For each window b that has not already been marked *dense* in previous iterations, we sample $\frac{|\mathcal{A}_{\text{Bad}}|\log^2(n)}{\deg_g}$ $\mathcal{A}_{\text{Bad}}$-windows a at random and query $\text{ED}(b, a)$. We say that b is *dense* if at least $\frac{1}{2}\log^2(n)$ of the samples are within edit distance τ.

If b is dense, we query its entire $\mathcal{N}^{\mathcal{A}_{\text{Bad}},\tau}(b), \mathcal{N}^{\mathcal{B}_\tau,2\tau}(b)$ neighborhoods. We (implicitly) add edges with cost 3τ for every pair in $\mathcal{N}^{\mathcal{A}_{\text{Bad}},\tau}(b) \times \mathcal{N}^{\mathcal{B}_\tau,2\tau}(b)$, and move the windows in $\mathcal{N}^{\mathcal{A}_{\text{Bad}},\tau}(b)$ to $\mathcal{A}_{\text{Covered}}$.

If the number of $\mathcal{A}_{\text{Bad}}$ windows becomes at most $t_\tau^{1/2}$ at any point, we exhaustively find all their neighbors in $\mathcal{B}_\tau$ and move them to $\mathcal{A}_{\text{Covered}}$.

Approximation. By triangle inequality, every pair of windows in $\mathcal{N}^{\mathcal{A}_{\text{Bad}},\tau}(b) \times \mathcal{N}^{\mathcal{B}_\tau,2\tau}(b)$ has edit distance at most 3τ. Notice also that by triangle inequality $\mathcal{N}^{\mathcal{B}_\tau,\tau}(\mathcal{N}^{\mathcal{A}_{\text{Bad}},\tau}(b)) \subseteq \mathcal{N}^{\mathcal{B}_\tau,2\tau}(b)$, i.e. we have discovered all the $(\mathcal{B}_\tau, \tau)$-neighbors of all the $\mathcal{A}_{\text{Covered}}$-windows.

Complexity: We maintain the bound that at the beginning of the g-th iteration, $|\mathcal{A}_{\text{Bad}}| = O(t_\tau^{1-(g-1)}) = O(t_\tau^{1/2}\deg_g)$. Hence, similarly to Step A, we spend $\tilde{O}(t_\tau^{1/2})$ queries for estimating the degree of each $b \in \mathcal{B}_\tau$, for a total of $\tilde{O}(t_\tau^{3/2})$.

Every time we discover a dense b, we query its edit distance to $\le t + t_\tau$ windows, and decrease by $\Omega(\deg_g)$ the number of remaining $\mathcal{A}_{\text{Bad}}$-windows. Recall that we start the g-th iteration with at most $O(t_\tau^{1-(g-1)\epsilon}) = O(t_\tau^{1/2}\deg_g)$ $\mathcal{A}_{\text{Bad}}$-windows. Hence in total this step requires $O((t + t_\tau)\cdot t_\tau^{1/2}) = O(t_\tau^{3/2})$ queries.

The sparsification phase: iterating over Step B-g. We iteratively apply Step B-g $O(1/\epsilon)$ times. At the end of the g-th iteration, every remaining $\mathcal{B}_\tau$-window has at most $\deg_g = t_\tau^{1/2-(g-1)\epsilon}$ remaining $(\mathcal{A}_{\text{Bad}}, \tau)$-neighbors. Hence the total number of τ-close pairs in $\mathcal{B}_\tau \times \mathcal{A}_{\text{Bad}}$ is $t_\tau^{3/2-(g-1)\epsilon}$. Since every $\mathcal{A}_{\text{Bad}}$ window has $\Omega(t_\tau^{1/2+\epsilon})$ $(\mathcal{B}_\tau, \tau)$-neighbors[4], we have that $|\mathcal{A}_{\text{Bad}}| = O(t_\tau^{1-(g)\epsilon})$.

Step C. Sparse windows. We process the $\mathcal{A}_{\text{Sparse}}$-windows as in the sparse case in Section 5 (for completeness, we spell out the details below). This algorithm is somewhat simpler than Section 5 since we already determined in advance which windows are sparse and which are dense.

Intervals (first iteration): Consider a partition of $[n]$ into $t_\tau^{1/2+2\epsilon}$ contiguous intervals of length $n/t_\tau^{1/2+2\epsilon} \le t_\tau^{1/2-2\epsilon}\cdot d$. For A we define the *$\mathcal{A}$-interval* $I_{\mathcal{A}}$ corresponding to interval $I \subset [n]$ as the set of $\le t_\tau^{1/2-2\epsilon}$ windows with indices in I. Therefore, for A-windows they are either entirely contained in the interval or don't intersect it.

[4]Notice that the number of remaining neighbors for $a \in \mathcal{A}_{\text{Bad}}$ does not change during the run of the sparsification phase, since once any of a's neighbors is declared *dense*, we move a to $\mathcal{A}_{\text{Covered}}$.

For B, we let I/ϵ denote a $1/\epsilon$-factor expansion of I (i.e. the interval of length $|I|/\epsilon$ centered at I)[5]. We define the $\mathcal{B}^\tau$-interval $I_{\mathcal{B}^\tau}$ to be the set of windows that intersect I/ϵ. When clear from context we sometimes just call $I_{\mathcal{A}}, I_{\mathcal{B}^\tau}$ intervals.

Sparse subroutine (first iteration): For each interval $I_{\mathcal{A}}$, if at most $\log^2(n)$ of its windows are sparse, we simply query their entire $(\mathcal{B}^\tau, \tau)$-neighborhoods. Otherwise, we sample a random set $S(I_{\mathcal{A}})$ of $\log^2(n)$ windows from $I_{\mathcal{A}} \cap \mathcal{A}_{\text{SPARSE}}$. For every window in $S(I_{\mathcal{A}})$, we query its entire $(\mathcal{B}^\tau, \tau)$-neighborhood using t_τ queries. For each interval $I_{\mathcal{A}}$ we record the union of all $\tilde{O}(t_\tau^{1/2+\epsilon})$ intervals $\widehat{I_{\mathcal{B}^\tau}}$ that contain any $(\mathcal{B}^\tau, \tau)$-neighbors of any of the sparse samples $a \in S(I_{\mathcal{A}})$. We call these $\mathcal{B}$-windows the *relevant windows* for the windows in $I_{\mathcal{A}}$. We henceforth no longer look to match windows from $I_{\mathcal{A}}$ to irrelevant $\mathcal{B}$-windows. Note that in a low-skew mapping, if at least one of the samples is matched, then windows in $I_{\mathcal{A}}$ cannot be mapped to any irrelevant $\mathcal{B}$-windows under that mapping.

Approximation (first iteration): Recall that by Lemma 5.1 and Lemma 5.2, there is a low-skew monotone mapping that approximates the optimal transformation to within $(1 + O(\epsilon))$-factor. For any low-skew monotone mapping μ, the entire interval $I_{\mathcal{A}}$ is mapped to a single $\mathcal{B}^\tau$-interval $I_{\mathcal{B}^\tau}$. Suppose that $(1-\epsilon)$-fraction of the sparse windows in $I_{\mathcal{A}}$ are mapped to $\mathcal{B}^\tau$-windows (or $\perp$) of distance greater than τ. Then we can safely discard the τ-edges for the remaining ϵ-fraction of sparse windows with negligible loss in approximation factor. Hence in total we pay only $(1 + O(\epsilon))$-factor in approximation for sparse windows. Otherwise, w.h.p. at least one of the samples has a $(\mathcal{B}^\tau, \tau)$-neighbor in $I_{\mathcal{B}^\tau}$.

Complexity (first iteration): Each sparse $\mathcal{A}$-window has only $\tilde{O}\left(t_\tau^{1/2+\epsilon} \cdot t_\tau^{1/2-2\epsilon}\right) = \tilde{O}(t_\tau^{1-\epsilon})$ relevant windows. Since there are $t_\tau^{1/2+2\epsilon}$ $\mathcal{A}$-intervals, we spend use a total of $\tilde{O}(t_\tau^{3/2+\epsilon})$ queries.

Recursion. We recurse on the sparse subroutine, with the following modifications for the ℓ-th iteration.

- We increase the number of intervals to $t_\tau^{1/2+(\ell+2)\epsilon}$, and their size decreases accordingly to $\tilde{O}(t_\tau^{1/2-(\ell+2)\epsilon})$.
- For each sparse sample, we only compute the restriction of its $(\mathcal{B}^\tau, \tau)$-neighborhood to relevant windows. Hence we only spend $\tilde{O}\left(t_\tau^{1-\ell\epsilon}\right)$ queries for each sample, or a total of $\tilde{O}\left(t_\tau^{3/2+\epsilon}\right)$ queries across all intervals.
- The relevant windows for the next level of recursion are a (strict) subset of the relevant windows in the current level.

The recursion continues until each interval has less than $\log^2(n)$ sparse windows, after which we can simply query the distance of every remaining sparse window to all its relevant $\mathcal{B}_\tau$-windows.

Completing the proof of Theorem 6.1. As we argued above, the algorithm finds a $(3 + \epsilon)$-approximation using $\tilde{O}(t_\tau^{3/2+\epsilon})$ queries. Taking ϵ to be slightly sub-constant completes the proof of Theorem 6.1. □

[5]For example, if $I = [20, 30]$ then its 3-expansion is $[10, 40]$.

7 HARDNESS

In this section we formalize, in the context of (approximate) edit distance, the folklore intuition (based on [48]) that polynomial preprocessing can not circumvent fine-grained complexity lower bounds. In Subsection 7.1 we show that known fine-grained complexity hardness results for exact edit distance and related problems extend to accommodate polynomial preprocessing.

In Subsection 7.2 we consider the problem of edit distance approximation. There are essentially no conditional hardness results for this problem, and in fact recent work obtained a truly-subquadratic constant factor approximation algorithm [22]. Improving this factor, and in particular obtaining a truly-subquadratic $(1 + \epsilon)$-approximation factor, is perhaps the most important open problem in this area. There are evidences that providing $1 + o(1)$-factor approximation might be hard, as it implies new circuit lower bounds [1]. Theorem 7.4 shows that essentially any approximation factor that is obtained with polynomial preprocessing can also be obtained without it. Note that this holds unconditionally, even if (BP)-SETH is false.

7.1 SETH-Hardness of Exact String Alignment with Preprocessing

The Strong Exponential Time Hypothesis is an (extreme) strengthening of $\mathbb{P} \neq \mathbb{NP}$ postulating that k-SAT on n variables requires $2^{(1-\delta_k)n}$ time. Building on [3], we can prove our hardness based on the milder BP-SETH which replaces k-CNF with a branching program:

Hypothesis 1 (BP-SETH). Given a branching program over n variables of width W and length T such that $T^W = 2^{o(n)}$, deciding whether it has a satisfying assignment requires time $2^{(1-o(1))n}$ time.

THEOREM 7.1 (BP-SETH HARDNESS). *Unless BP-SETH is false, there is no algorithm that preprocesses two input strings in polynomial time and then computes their (edit distance / longest common subsequence / dynamic time warping) in truly-subquadratic time.*

Remark. We remark that unlike with k-SAT, it is plausible that the brute-force algorithm for BP-SAT is optimal to within $\text{poly}(n)$ factors, and in fact better algorithms would imply new circuit lower bounds ([3] and references therein). Under a corresponding strengthening of BP-SETH one can show that string alignment with preprocessing requires $N^2/\text{polylog}(N)$ time.

The proof of Theorem 7.1 builds on *alignment gadgets* and *normalized vector gadgets (NVG)* from previous works on SETH and BP-SETH hardness of string alignment [3, 11, 20]. Each NVG represents a half-assignment to the branching program, and the alignment gadgets define a composition of the NVGs into two long strings. Here we deviate from typical SETH-hardness proofs of sequence similarity, and use a divide-and-conquer approach of [48] to construct two larger sets of shorter strings. This allows us to reuse the preprocessing of each shorter string when we compare every pair to look for a satisfying assignment (aka a satisfying pair of half-assignments).

Below we use dist() to refer to the distance under the relevant similarity measure (edit distance / longest common subsequence / dynamic time warping); for longest common subsequence

we use the "co-LCS" (edit-distance-without-substitutions) distance $\mathrm{dist}(A,B) := n - \mathrm{LCS}(A,B)$.

Normalized Vector Gadgets. Given a BP φ of width W and length T, *normalized vector gadgets* (NVG) map half assignments $a, b \in \{0,1\}^{n/2}$ into strings such that:

$$\mathrm{dist}(\mathrm{NVG}_A(a), \mathrm{NVG}_B(b)) = \begin{cases} c_T & \text{if assignment } (a \circ b) \text{ satisfies } \varphi; \\ c_F & \text{otherwise} \end{cases},$$

where $c_T < c_F$ are integers that depend on W, T.

Lemma 7.2 (Normalized Vector Gadgets [3]). *Given a BP of width W and length T, we can construct NVGs of length $T^{O(\log(W))}$) for all half assignments $a, b \in \{0,1\}^{n/2}$ in time $2^{n/2} \cdot T^{O(\log(W))}$).*

Alignment Gadgets. Consider two ordered sets of strings $\mathcal{A}, \mathcal{B}$ of cardinalities $n_A < n_B$, respectively. An alignment μ is a monotone partial mapping from $\mathcal{A}$ to $\mathcal{B} \cup \{\perp\}$. An alignment μ is *structured* if it maps the i-th string in $\mathcal{A}$ to the $i + \Delta$ string in $\mathcal{B}$ for some fixed shift Δ and for all $a \in \mathcal{A}$.

The cost of a mapping μ is defined by:

$$\mathrm{cost}(\mu, \mathcal{A}, \mathcal{B}) := \sum_{a \in \mathcal{A}} \mathrm{dist}(a, \mu(a)).$$

Here $\mathrm{dist}(a, \perp) := \max_{a' \in \mathcal{A}, b \in \mathcal{B}} \mathrm{dist}(a', b)$.

An *alignment gadget* is a mapping from $\mathcal{A}, \mathcal{B}$ into respective strings $\mathrm{GA}_A(\mathcal{A}), \mathrm{GA}_B(\mathcal{B})$ such that for some parameter $c_{\mathcal{GA}} = c(n_A, n_B)$:

$$\min_{\text{alignment } \mu} \mathrm{cost}(\mu, \mathcal{A}, \mathcal{B}) \le \mathrm{dist}(\mathrm{GA}_A(\mathcal{A}), \mathrm{GA}_B(\mathcal{B})) + c_{\mathcal{GA}}$$
$$\le \min_{\text{structured alignment } \mu} \mathrm{cost}(\mu, \mathcal{A}, \mathcal{B}). \quad (5)$$

Lemma 7.3 (Alignment gadgets [20]). *Edit distance, LCS (with binary alphabet), and Dynamic Time Warping admit alignment gadgets that can be computed in linear time.*

Completing the Proof of BP-SETH-Hardness.

Proof of Theorem 7.1. Suppose that we have an algorithm that computes dist() for strings of length N with preprocessing time $O(N^t)$ and query time $O(N^{2-\epsilon})$. Given a BP over $n = 2\log_2(N)$ variables, we construct all its normalized vector gadgets in near-linear time as in Lemma 7.2.

We partition the $\mathcal{A}$-NVGs into $2^{(1-1/t)n}$ subsets $\mathcal{A}_1, \ldots \mathcal{A}_{2^{(1-1/t)n}}$ of size $2^{n/t}$ each (and likewise for $\mathcal{B}$). For each subset $\mathcal{A}_i$, we construct its alignment gadget A_i of size $\tilde{O}(N^{1/t})$. For $\mathcal{B}_j$, let B_j be constructed by the alignment gadget for the set repeated twice. If no pair of half-assignments corresponding to $\mathcal{A}_i \times \mathcal{B}_j$ satisfies the BP, then every pair of NVGs is at distance c_F, and by (5) the distance of A_i, B_j will be $d_F := 2^{n/t} c_F - c_{\mathcal{GA}}$. If there is a satisfying pair, then the structured alignment that matches the corresponding NVGs will have cost at most

$$d_T := c_T + (2^{n/t} - 1)c_F - c_{\mathcal{GA}} < d_F.$$

We preprocess all the strings in total time $\tilde{O}\left(N^{1-1/t} \cdot (N^{1/t})^t\right) = \tilde{O}(N^{2-1/t})$. Finally, we compute the distance between all $(N^{1-1/t})^2$ pairs in time $O\left(N^{2-2/t} \cdot (N^{1/t})^{2-\epsilon}\right) = O\left(N^{2-2\epsilon/t}\right)$. The BP is satisfiable iff at least one of the pairs is at distance at most d_T. □

7.2 Preprocessing Doesn't Help for Approximate ED in Truly-Subquadratic Time

Theorem 7.4 (Hardness of Approximation). *If there is an α-approximation algorithm for edit distance that runs in polynomial preprocessing time and truly-subquadratic query time, then there is an $(\alpha + o(1))$-approximation algorithm that runs in truly-subquadratic time with no preprocessing.*

The proof combines the divide-and-conquer steps from our approximate edit distance algorithm (Section 5) with that of [48] (see also last step in the proof of Theorem 7.1).

Proof. Suppose that there exists an algorithm that computes an α-approximation of edit distance using $O(n^t)$-preprocessing and $O(n^{2-\epsilon})$-query time. First, we assume wlog that the true edit distance is $k = \omega(n^{1-1/2t})$, otherwise we can solve the problem in time $O(n^{2-1/t})$ using the algorithm of [35]. In particular, we can henceforth neglect additive errors of $O(n^{1-1/2t})$.

Using the notation of Section 5.2, we decompose the strings into windows with base width $d := n^{1/t}$. The A-windows have no overlap, and for the B-windows we consider $\mathcal{B}^\tau$ for $\tau = n^{-1/2t}$. Hence we have $\tilde{O}(n^{1-1/t})$ $\mathcal{A}$-windows and $\tilde{O}(n^{1-1/2t})$ $\mathcal{B}$-windows, all of length $O(n^{1/t})$.

We preprocess all the windows in time

$$\tilde{O}\left(n^{1-1/2t} \cdot (n^{1/t})^t\right) = \tilde{O}\left(n^{2-1/2t}\right).$$

We then run the α-approximate edit distance algorithm on pairs of windows. By the argument of [47], it suffices to only compute the distances between pairs of windows whose starting points are within $\pm k$ far apart. In particular for every $\mathcal{A}$-window, we only need to compute the edit distance to $\tilde{O}(n^{1-1/t})$ $\mathcal{B}$-windows. In total we spend $\tilde{O}\left((n^{1-1/t})^2 \cdot (n^{1/t})^{2-\epsilon}\right) = \tilde{O}\left(n^{2-\epsilon/t}\right)$ time on this phase.

Given the α-approximate window-window distance estimates, we aggregate them in time $\tilde{O}(n^{2-2/t})$ using Lemma 5.3. Thus we obtain an α-approximation to the optimal window-compatible matching, which by Lemma 5.1, is an $(\alpha + o(1))$-approximation to the edit distance. □

REFERENCES

[1] Amir Abboud and Arturs Backurs. 2017. Towards Hardness of Approximation for Polynomial Time Problems. In *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*. 11:1–11:26. https://doi.org/10.4230/LIPIcs.ITCS.2017.11

[2] Amir Abboud and Karl Bringmann. 2018. Tighter Connections Between Formula-SAT and Shaving Logs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. 8:1–8:18. https://doi.org/10.4230/LIPIcs.ICALP.2018.8

[3] 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 *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 375–388. https://doi.org/10.1145/2897518.2897653

[4] Amir Abboud, Aviad Rubinstein, and R. Ryan Williams. 2017. Distributed PCP Theorems for Hardness of Approximation in P. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. 25–36. https://doi.org/10.1109/FOCS.2017.12

[5] Amir Abboud and Virginia Vassilevska Williams. 2019. Personal communication.

[6] Alexandr Andoni. 2019. Simpler Constant-Factor Approximation to Edit Distance Problems. In preparation.

[7] Alexandr Andoni, Michel Deza, Anupam Gupta, Piotr Indyk, and Sofya Raskhodnikova. 2003. Lower bounds for embedding edit distance into normed spaces. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete*

Algorithms, January 12-14, 2003, Baltimore, Maryland, USA. 523–526. http://dl.acm.org/citation.cfm?id=644108.644196

[8] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. 2010. Polylogarithmic Approximation for Edit Distance and the Asymmetric Query Complexity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. 377–386. https://doi.org/10.1109/FOCS.2010.43

[9] Alexandr Andoni and Huy L. Nguyen. 2010. Near-Optimal Sublinear Time Algorithms for Ulam Distance. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. 76–86. https://doi.org/10.1137/1.9781611973075.8

[10] Alexandr Andoni and Krzysztof Onak. 2012. Approximating Edit Distance in Near-Linear Time. *SIAM J. Comput.* 41, 6 (2012), 1635–1648. https://doi.org/10.1137/090767182

[11] Arturs Backurs and Piotr Indyk. 2018. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). *SIAM J. Comput.* 47, 3 (2018), 1087–1097. https://doi.org/10.1137/15M1053128

[12] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. 2004. Approximating Edit Distance Efficiently. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*. 550–559. https://doi.org/10.1109/FOCS.2004.14

[13] 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 *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*. 316–324. https://doi.org/10.1145/780542.780590

[14] Tugkan Batu, Funda Ergün, and Süleyman Cenk Sahinalp. 2006. Oblivious string embeddings and edit distance approximations. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*. 792–801. http://dl.acm.org/citation.cfm?id=1109557.1109644

[15] Djamal Belazzougui and Qin Zhang. 2016. Edit Distance: Sketching, Streaming, and Document Exchange. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. 51–60. https://doi.org/10.1109/FOCS.2016.15

[16] Omri Ben-Eliezer, Clément L. Canonne, Shoham Letzter, and Erik Waingarten. 2019. Finding monotone patterns in sublinear time. *CoRR* abs/1910.01749 (2019). arXiv:1910.01749 http://arxiv.org/abs/1910.01749

[17] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. 2018. Approximating Edit Distance in Truly Subquadratic Time: Quantum and MapReduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1170–1189. https://doi.org/10.1137/1.9781611975031.76

[18] Joshua Brakensiek, Venkatesan Guruswami, and Samuel Zbarsky. 2016. Efficient Low-redundancy Codes for Correcting Multiple Deletions. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '16)*.

[19] Joshua Brakensiek and Aviad Rubinstein. 2019. Constant-factor approximation of near-linear edit distance in near-linear time. *CoRR* abs/1904.05390 (2019). arXiv:1904.05390 http://arxiv.org/abs/1904.05390

[20] Karl Bringmann and Marvin Künnemann. 2015. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. 79–97. https://doi.org/10.1109/FOCS.2015.15

[21] Karl Bringmann and Marvin Künnemann. 2018. Multivariate Fine-Grained Complexity of Longest Common Subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1216–1235. https://doi.org/10.1137/1.9781611975031.79

[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 *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*. 979–990. https://doi.org/10.1109/FOCS.2018.00096

[23] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. 2016. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 712–725.

[24] Moses Charikar, Ofir Geri, Michael P. Kim, and William Kuszmaul. 2018. On Estimating Edit Distance: Alignment, Dimension Reduction, and Embeddings. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. 34:1–34:14. https://doi.org/10.4230/LIPIcs.ICALP.2018.34

[25] Moses Charikar and Robert Krauthgamer. 2006. Embedding the Ulam metric into l_1. *Theory of Computing* 2, 11 (2006), 207–224. https://doi.org/10.4086/toc.2006.v002a011

[26] Lijie Chen, Shafi Goldwasser, Kaifeng Lyu, Guy N. Rothblum, and Aviad Rubinstein. 2019. Fine-grained Complexity Meets IP = PSPACE. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. 1–20. https://doi.org/10.1137/1.9781611975482.1

[27] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. 2019. Sublinear Algorithms for Gap Edit Distance. *FOCS* abs/1910.00901 (2019). arXiv:1910.00901 http://arxiv.org/abs/1910.00901

[28] Bernhard Haeupler. 2019. Optimal Document Exchange and New Codes for Insertions and Deletions. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS*.

[29] Bernhard Haeupler, Aviad Rubinstein, and Amirbehshad Shahrasbi. 2019. Near-linear time insertion-deletion codes and (1+ϵ)-approximating edit distance via indexing. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. 697–708.

[30] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. 2019. Approximating LCS in Linear Time: Beating the $\sqrt{n}$ Barrier. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. 1181–1200. https://doi.org/10.1137/1.9781611975482.72

[31] Piotr Indyk. 2004. Approximate Nearest Neighbor under edit distance via product metrics. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. 646–650. http://dl.acm.org/citation.cfm?id=982792.982889

[32] Guy Jacobson and Kiem-Phong Vo. 1992. Heaviest Increasing/Common Subsequence Problems. In *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*. 52–66. https://doi.org/10.1007/3-540-56024-6_5

[33] Hossein Jowhari. 2012. Efficient Communication Protocols for Deciding Edit Distance. In *Proceedings of the 20th Annual European Conference on Algorithms* (Ljubljana, Slovenia) *(ESA'12)*. 648–658.

[34] Michal Koucký and Michael E. Saks. 2019. Constant factor approximations to edit distance on far input pairs in nearly linear time. *CoRR* abs/1904.05459 (2019). arXiv:1904.05459 http://arxiv.org/abs/1904.05459

[35] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. 1998. Incremental String Comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582. https://doi.org/10.1137/S0097539794264810

[36] Gad M. Landau and Uzi Vishkin. 1988. Fast String Matching with k Differences. *J. Comput. Syst. Sci.* 37, 1 (1988), 63–78. https://doi.org/10.1016/0022-0000(88)90045-1

[37] VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

[38] Wei Lu, Xiaoyong Du, Marios Hadjieleftheriou, and Beng Chin Ooi. 2014. Efficiently Supporting Edit Distance Based String Similarity Search Using B $ˆ+$-Trees. *IEEE Trans. Knowl. Data Eng.* 26, 12 (2014), 2983–2996. https://doi.org/10.1109/TKDE.2014.2309131

[39] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446

[40] Ilan Newman, Yuri Rabinovich, Deepak Rajendraprasad, and Christian Sohler. 2019. Testing for forbidden order patterns in an array. *Random Struct. Algorithms* 55, 2 (2019), 402–426. https://doi.org/10.1002/rsa.20840

[41] Rafail Ostrovsky and Yuval Rabani. 2007. Low distortion embeddings for edit distance. *J. ACM* 54, 5 (2007), 23. https://doi.org/10.1145/1284320.1284322

[42] Aviad Rubinstein. 2018. Approximating Edit Distance. https://theorydish.blog/2018/07/20/approximating-edit-distance/.

[43] Aviad Rubinstein. 2018. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*. 1260–1268. https://doi.org/10.1145/3188745.3188916

[44] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. 2019. Approximation Algorithms for LCS and LIS with Truly Improved Running Times. In *FOCS 2019*. To appear.

[45] Aviad Rubinstein and Zhao Song. 2019. Reducing approximate Longest Common Subsequence to approximate Edit Distance. *CoRR* abs/1904.05451 (2019). arXiv:1904.05451 http://arxiv.org/abs/1904.05451

[46] Michael E. Saks and C. Seshadhri. 2017. Estimating the Longest Increasing Sequence in Polylogarithmic Time. *SIAM J. Comput.* 46, 2 (2017), 774–823. https://doi.org/10.1137/130942152

[47] Esko Ukkonen. 1985. Algorithms for Approximate String Matching. *Information and Control* 64, 1-3 (1985), 100–118. https://doi.org/10.1016/S0019-9958(85)80046-2

[48] Virginia Vassilevska Williams and R. Ryan Williams. 2018. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *J. ACM* 65, 5 (2018), 27:1–27:38. https://doi.org/10.1145/3186893