Simple Constant-Factor Approximation to Edit Distance

Alexandr Andoni*
Columbia University
andoni@cs.columbia.edu

August 19, 2020

Abstract

We show a simple algorithm for estimating edit distance between two strings in strongly sub-quadratic time, up to $3+\epsilon$ approximation. The recent couple years have seen a "new generation" of edit distance algorithms, achieving constant factor approximation, and this write-up aims to provide the simplest point of entry illustrating the central ideas used. We note that, when the edit distance is close to linear in the strings' length, our algorithm matches the runtime of the fastest such algorithm for $3+\epsilon$ approximation from the independent work of [GRS20].

1 Introduction

Edit distance is a classic distance measure between sequences that takes into account the (mis)alignment of strings, and is defined for two strings of length n over some alphabet Σ, as the number of insertions/deletions/substitutions of characters to transform one string into the other. It is of key importance in several applied fields. Within TCS, edit distance is also an illustrious example of the dynamic programming technique, with a classic quadratic run-time solution [WF74]. It has since proven to be a great challenge in a central theme in TCS: to improve the run-time from polynomial to close(r) to linear. Despite significant research over many decades, the running time has so far been improved only slightly, to $O(n^2/\log^2 n)$ [MP80], which remains the fastest algorithm known to date. In fact a strongly sub-quadratic algorithm would disprove the Strong Exponential Time Hypothesis (SETH) [BI15] (and even more plausible conjectures [AHWW16]).

The breakthrough result of [CDG+18] obtained a constant factor approximation in strongly sub-quadratic time of $O(n^{1.714})$ (see also the contemporary quantum algorithm [BEG+18]). This has led to a number of follow-ups (independent of this paper[1]) improving the approximation–runtime trade-off. When allowing *additive* approximation, [KS19, BR19] obtained near-linear runtime. [AS20] obtained a constant factor approximation in near-linear time without further restrictions. For the orthogonal goal of obtaining the smallest possible approximation (in truly sub-quadratic time), [GRS20] design a $\tilde{O}(n^{1.6})$ time algorithm for $3+\epsilon$ approximation. In the special regime when the distance is $\Omega(n)$ (i.e., near-maximal additive approximation), [RSSS19] obtain a $3-\Omega(1)$ factor

*Research supported in part by the Simons Foundation (#491119 to Alexandr Andoni), and NSF (CCF-1617955, CCF-1740833).

[1] A more elaborate version of this write-up was first distributed privately in October 2018. See that version here http://www.cs.columbia.edu/~andoni/papers/edit/

approximation (although there are less reasons to believe that such approximation is achievable when the output distance is $o(n)$).

The goal of this paper is to present the *simplest* algorithm for constant approximation and strongly subquadratic time. Besides providing an easier entry point into the recent edit distance results, we hope the paper highlights the following concrete open question: obtain the fastest algorithm achieving $3+\epsilon$ approximation.

Theorem 1.1 (Main theorem, Section 2)**.** *For any $\epsilon > 0, \beta > 1$, we can estimate edit distance between two strings of length n in time $\tilde{O}_\epsilon(n^{1.6}\beta^{1.4})$, achieving $(3+\epsilon)$-factor and $\pm n/\beta$ additive approximation. When further combined with the exact $O(n+(n/\beta)^2)$-time algorithm [Ukk85, Mye86], we obtain $O_\epsilon(n^{1.765})$ time.*

We remark that we can improve the runtime to $\tilde{O}(n^{1.6}\beta^{0.6})$ and $O(n^{1.693})$ overall by using a slightly stronger edit distance subroutine instead of the standard (exact) edit distance algorithm, used by the above theorem on substrings of x/y (see technical overview below). In particular, we can exploit the following version of the text searching problem: given a pattern $p \in \Sigma^w$, a text $T \in \Sigma^n$, and a length $l = O(w)$, compute distance from p to each substring of T of length l. Indeed, a modification of the standard edit distance dynamic programming algorithm gives a $(1+\epsilon)$-approximate solution to this problem, with a $\tilde{O}_\epsilon(nw)$ time solution. See details in Section 3.

We also note that our runtime matches the best known runtime of $\tilde{O}(n^{1.6})$, for the same approximation, from the independent work of [GRS20], when β is small. To remove the dependence on β in their algorithm, a key ingredient is also a different subroutine substituting the standard (exact) edit distance algorithm. That subroutine is able to, after some preprocessing, to compute $\mathrm{ed}(x,y)$ in time $\tilde{O}(n^2/\beta^2)$ when $\mathrm{ed}(x,y) \le n/\beta$ (and is applied to substrings of x/y).

In either case, it remains a challenge to obtain a runtime below $n^{1.6}$, i.e., even for small β.

1.1 Related Past Work

Subquadratic algorithms for edit distance have been studied for over a couple decades. One can obtain linear-time algorithm with $\sqrt{n}$ approximation from the exact algorithm of [Ukk85, Mye86], which runs in time $O(n + (n/\beta)^2)$, when edit distance is at most n/β. Focusing explicitly on approximate algorithms in near-linear time, researchers obtained $n^{3/7}$ approximation in [BJKK04], $n^{1/3+o(1)}$ in [BES06], $2^{\tilde{O}(\sqrt{\log n})}$ in [AO12], and finally $(\log n)^{O(1/\epsilon)}$ in $O(n^{1+\epsilon})$ time in [AKO10].

Sublinear-time algorithms also exist, starting with the algorithm of [BEK+03] that achieves n^ϵ approximation when $\beta = O(1)$ is large. Recent results showed how to distinguish distance k vs $\Omega(k^2)$ first in $\tilde{O}(n/k + k^3)$ time [GKS19] and most recently in $\tilde{O}(n/k + k^2)$ time [KS20].

1.2 Algorithm synopsis and notation

First, for simplicity, we index everything from zero. Hence, $[n] = \{0, 1, \ldots n-1\}$. Also $x[i:j]$ is the string starting at position i until position $j-1$.

As with most of the approximate algorithms for the problem, the overall approach is to partition the two strings into blocks of length w (think $w = n^{0.2}$), and perform a modified dynamic programming on these blocks. In particular, we partition x into $b = n/w$ blocks of length w. Define $I = w \cdot [b]$ to be the starting positions, and $X_i = x[i : i+w]$ for $i \in I$. We also split y into blocks of length w, though we need to consider overlapping blocks, especially when $\beta \gg 1$. For

$\Delta = \max\{1, \epsilon w/\beta\}$, the starting positions into y are from the set $J = \Delta \cdot [n/\Delta]$; resulting blocks are called Y_j. For this intuituve overview, assume that $\beta = O(1)$, and hence $|I|, |J| = O(b)$.

We can now run a dynamic programming on the "blocks" by building a graph on node-set $(I \cup \{n\}) \times (J \cup \{n\})$ with diagonal edges $(i, j) \to (i + w, j + w)$ with cost $\mathrm{ed}(X_i, Y_j)$ corresponding to matching those blocks, as well as horizontal/vertical edges (corresponding to deleting contiguous chunks from x/y). Then output is the shortest path from $(0, 0)$ to (n, n), which corresponds to some optimal matching $i \mapsto j_i$ from the x-blocks to the y-blocks.[2] A standard claim is that this gives an approximation to $\mathrm{ed}(x, y)$, up to an additive n/β term (due to rounding of the j's to J). To compute the graph edges, we need to perform $\approx b^2 = (n/w)^2$ ed-computations $\mathrm{ed}(X_i, Y_j)$, each taking $O(w^2)$ time—i.e., no time gains so far. The main goal is then to reduce the number of ed-computations to $\ll b^2$, noting that the shortest path computation takes only $\approx b^2 \ll n^2$ time.

There are two central ideas to reduce the number of ed-calculations. The first one is triangle inequality of $\mathrm{ed}(\cdot, \cdot)$ metric (idea originally due to [BEG+18] and [CDG+18]). In our algorithm, we sample $k = \sqrt{b}$ y-blocks, termed "centers", and compute the distance from the centers to all other y-blocks. Then, for each x-block X_i, if Y_{π_i} is the closest center and $d_i = \mathrm{ed}(X_i, Y_{\pi_i})$, then we can upper bound distance the $\mathrm{ed}(X_i, Y_j)$ to an arbitrary block Y_j as $d_i + \mathrm{ed}(Y_{\pi_i}, Y_j)$ by triangle inequality. Computing all these estimates takes only $2bk = O((n/w)^{1.5})$ ed-calculations.

This upper bound is good enough when the distance from X_i to its matched Y_{j_i} (in the optimal edit-distance alignment on blocks) is d_i or more. Indeed, calling the latter $c_i = \mathrm{ed}(X_i, Y_{j_i})$, when $c_i \geq d_i$, our upper bound becomes $\leq d_i + d_i + c_i \leq 3c_i$, i.e., a 3-approximation.

Obviously the above estimate can be a gross overestimate, which leads us to the second idea (originally due to [CDG+18]). For a fixed i, suppose that $c_i < d_i$, which is usually referred to as the "sparse" case. Given that we sample k random center, there can only be $\approx b/k$ y-blocks j where $\mathrm{ed}(X_i, Y_j) < d_i$, and our new goal is to identify them. As we don't actually know which are those j's without performing the ed-calculation, we still need $\approx b$ ed-computations for that i. However this calculation will help other sparse indeces! Suppose we've computed the set S_i of j's where $\mathrm{ed}(X_i, Y_j) < d_i$, i.e., the set containing the (unknown) optimal match j_i. Now suppose that X_{i+1} also happens to be sparse. Its optimal match j_{i+1} must be close to j_i, and hence to something from S_i. Hence we now only need to check $\mathrm{ed}(X_{i+1}, Y_j)$ for j in some small extension of S_i, for a total of $\approx b/k$ ed-computations only!

Indeed, a key lemma proves that there exists a near-optimal map $i \mapsto j_i$ that is Lipshitz, namely for any i, i', $|j_{i'} - j_i| \leq \frac{1}{\epsilon}|i - i'|$ (Lemma 2.3). Thus, even if X_{i+1} is not sparse, but only some X_{i+s} for $s \gg 1$, then we need to check only $S_i + \{0, \pm 1, \ldots \pm s/\epsilon\}$, for a total of $O_\epsilon(s \cdot b/k)$ ed-computations. These computations can be "charged" to the $s - 1$ indexes that were not sparse, still averaging $O_\epsilon(b/k)$ per index. Overall, we expect to use only extra $\approx b \cdot b/k = b^{1.5}$ ed-calculations per index to process the sparse indeces i.

A caveat in the above is that we don't know which indeces i are sparse (as we don't know the matching j_i or cost c_i). Therefore, rather than "transferring" set S_i to the next adjacent sparse index, we pick indeces i in a random order. In particular, we partition I into dyadic intervals and proceed in a top-bottom fashion in the corresponding binary tree. At the root we sample a few "anchors" $q \in I$, and compute their distance to each Y_j, and for each anchor q store a set S_q of candidates j_i (i.e., where distance is smaller than d_q). In a node corresponding to a subinterval $I' \subset I$, we sample a few anchors and, for each anchor $q \in I'$ compute the set S_q as before using the (extensions of the) sets $S_{q'}$ where q' are the anchors of the parent node. We highlight the

[2] Note that the classic edit distance algorithm corresponds to the case $w = 1$.

fact that the anchors q are sampled with probability proportional to the quantity d_q (intutively, intervals with a high d_q are more likely to be sparse). Even with this bias, it can happen that some sub-interval is composed of very few sparse indexes, in which case we show that the upper bound from the first idea is enough overall (i.e., we will not approximate well $\mathrm{ed}(X_i, Y_{j_i})$ for all i).

Finally, for the aforementioned near-optimal map lemma, we need to consider y-blocks of variable length. Let the set of lengths be $L = \{0\} \cup (w \pm \Delta \cdot E) \cap [\epsilon w, w/\epsilon]$ where E is the set of powers of $1+\epsilon$. The y-blocks are $Y_{j,l} = y[j : j+l]$ where $j \in J$ and $l \in L$. Note that $|L| \leq O(\log n)$. Let $p = |J \times L| = O(n/w \cdot \frac{\beta \log n}{\epsilon})$.

2 Main algorithm and analysis

Our algorithm relies on a subroutine to compute edit distance (exactly) between strings of length w, for which we use the standard $O(w^2)$ time algorithm. We remark that, in Section 3 we show how to get a slight runtime improvement by using a slightly stronger primitive instead. Also, as is done in other work, one could instead replace the exact edit distance computation by an approximate one, in a recursive fashion. The main theorem follows from the following theorem proven below:

Theorem 2.1. *For any $\epsilon > 0$, $\beta > 1$, there exists a randomized algorithm to estimate edit distance in time $\tilde{O}_\epsilon\left((\frac{n\beta}{w})^{3/2} \cdot t(w) + (\frac{n}{w})^2\beta\right)$, achieving multiplicative approximation $3+\epsilon$ and additive approximation n/β, where $t(w)$ is time to compute edit distance between two strings of length w.*

2.1 Algorithm description

At a high level, our algorithm constructs a graph corresponding the dynamic programming on w-length blocks. There are two phases of adding edges to the graph. After the two steps are completed, we perform a simple shortest distance computation in the resulting graph.

We build a grid graph on vertices $\bar{I} \times \bar{J}$ (just I, J with one extra index at the end). In particular, we have edges (i,j) to $(i+w, j)$ of cost w, as well as edges (i,j) to $(i, j+\Delta)$ of cost Δ. Most importantly, for each triple $(i,j,l) \in I \times J \times L$, we will add an edge (i,j) to $(i+w, j+l)$ with some cost which upper bounds the distance $\mathrm{ed}(X_i, Y_{j,l})$. In the first phase, the algorithm produces upper bounds for all the triples (i,j,l). The second phase will produce a more accurate upper bound for *some* of these edges, in which case we keep the lower cost.

Finally we add edges $(0,0)$ to $(0,j)$ for each $j \in S$, with zero cost.

Phase 1: dense. Fix k to be the number of y-centers, tbd (think $k \approx \sqrt{b}$). Now pick k random pairs $\pi = (j,l) \in J \times L$ (the centers), forming the set $C \subset J \times L$. For each $\pi \in C$, compute the edit distance between Y_π and every other Y_τ where $\tau \in J \times L$, and store it as $K_{\pi,\tau}$.

Now for each $i \in [b]$, compute the edit distance from X_i to Y_π for all $\pi \in C$. Let d_i be the minimal distance found, and π_i be the corresponding minimizing pair. Now add to the graph all edges corresponding to (i,j,l), with cost $d_i + K_{\pi_i,\tau}$ where $\tau = (j,l)$.

Phase 2: sparse. We build a binary tree of depth $O(\log b)$ as follows. Each node corresponds to a sub-interval of I, where the root corresponds to the entire I. For each node, starting from the root, we partition its interval into two and assign them to the two children. The tree has $|I|$ children corresponding to singletons. Now, for each node v, with the interval $U_v \subseteq I$, we sample

$h = O(\frac{\log n}{\epsilon})$ anchors at random from U_v, with probabilities proportional to d_q (with repetition): i.e., q is chosen with probability $d_q / \sum_{q' \in U_v} d_{q'}$. The sampled set of anchors is termed Q_v.

For the ensuing computations, we introduce a definition:

Definition 2.2. *Two triples $(q, j, l), (q', j', l') \in I \times J \times L$ are* compatible *if $|j - j'| \leq \frac{1}{\epsilon} \cdot |q - q'|$.*

For each top-level anchor $q \in I$, we compute the edit distance from X_q to each block Y_π for $\pi \in J \times L$. Let S_q be the set of pairs (j, l) such that the edit distance is less than d_q. For each anchor q at non-top level, whose parent r has anchors $Q_r \subset I$, we compute the set $\breve{S}_q \subseteq J \times L$ to be the set of pairs that are *compatible* with anything from $\cup_{q' \in Q_r} q' \times S_{q'}$. Then, we compute the edit distance from X_q to each Y_π for $\pi \in \breve{S}$, and store the set S_q of all pairs π where the distance is $< d_q$ (updating the corresponding costs in the graph).

Finally, we just compute the shortest path distance in the resulting graph, from $(0, 0)$ to (n, n).

2.2 Analysis: correctness

First, we note that, by the construction of the graph, if there exists a path of cost ζ, then there's an alignment between x and y with edit distance of cost at most ζ as well. Hence, our main task is to prove that there exists a path of cost at most $(3 + \epsilon) \operatorname{ed}(x, y)$.

The following lemma shows that there is path in our graph on blocks, of cost about $\operatorname{ed}(x, y)$.

Lemma 2.3. *Fix $\epsilon > 0$, as well as $1 \leq \beta \leq n$, and two strings x, y of length n, divisible by w. Let $I = w \cdot [n/w]$ and $J = \Delta \cdot [n/\Delta]$, where $\Delta = \epsilon w / \beta$. There's a matching between the x-blocks and y-blocks, described by triples (i, j_i, l_i) for $i \in I$, such that*[3]

$$\sum_{i \in I} \operatorname{ed}(X_i, Y_{j_i, l_i}) + (j_i - (j_{i-1} + l_{i-1})) \leq (1 + O(\epsilon)) \operatorname{ed}(x, y) + O(\epsilon n / \beta), \tag{1}$$

and (j_i, l_i) satisfy the following properties:

- *$(j_i, l_i) \in S \times L$;*
- *they are disjoint and in order: $j_i + l_i \leq j_{i+1}$;*
- *for any $i < i'$, the triples (i, j, l) and (i', j', l') are compatible: i.e., $j_{i'} - j_i \leq \frac{1}{\epsilon} \cdot (i' - i)$.*

We now fix the matching $(i, j_i, l_i)_{i \in I}$ from the above lemma. Note that the matching corresponds to a path in our graph, with the caveat that the edges $(i, j_i) \to (i + w, j_i + l_i)$ (corresponding to the $\operatorname{ed}(X_i, Y_{j_i, l_i})$ terms) may have higher weights. We show below that the total cost of these diagonal edges in our graph is upper bounded by $1 + O(\epsilon)$ times $\sum_{i \in I} \operatorname{ed}(X_i, Y_{j_i, l_i})$. We define $c_i = \operatorname{ed}(X_i, Y_{j_i, l_i})$.

In the first stage of the algorithm, for each $\tau \in J \times L$, we add edge corresponding to (i, τ), with cost $d_i + K_{\pi_i, \tau}$. When $\tau = (j_i, l_i)$, we have

$$K_{\pi_i, (j_i, l_i)} = \operatorname{ed}(Y_{\pi_i}, Y_{j_i, l_i}) \leq \operatorname{ed}(Y_{\pi_i}, X_i) + \operatorname{ed}(X_i, Y_{j_i, l_i}) = d_i + c_i$$

and hence the edge (i, j_i, l_i) has cost at most $d_i + K_{\pi_i, (j_i, l_i)} \leq 2d_i + c_i$ (it's "at most" because it may be replaced by lower-cost edge in the second phase). Note that for i's where $c_i \geq d_i$, the cost of the edge (i, j_i, l_i) becomes $\leq 3c_i$.

[3] By convention, $j_{-1} = l_{-1} = 0$.

The main challenge is to bound the cost of edges (i, j_i, l_i) when $c_i < d_i$, which is the purporse of the second phase of the algorithm. Define the set Z of "sparse" blocks to be the set of $i \in I$ where $c_i < d_i$. For a tree node v with sub-interval $U_v \subseteq I$, we call v to be *successful* if: 1) the set of anchors $Q_v \subset U_v$ includes at least one $q \in Z$, and 2) for some $q \in Q_v \cap Z \neq \emptyset$, we have that $(j_q, l_q) \in S_q$.

Lemma 2.4. *Consider any node v, and suppose all its ancestors are successful. If $\sum_{i \in U_v} c_i < \sum_{i \in U_v} (1-\epsilon) d_i$, then v is also successful with high probability.*

Proof. We first show that $Q_v \cap Z \neq \emptyset$, for which we employ the following claim.

Claim 2.5. *Consider m pairs of positive reals $c_i, d_i \geq 0$. Suppose that $\sum_i c_i < (1-\epsilon) \sum_i d_i$. Then, if we pick q with probability $d_q / \sum_i d_i$, we have that $\Pr_q[c_q < d_q] \geq \epsilon$.*

Proof. We have that:

$$1 - \epsilon > \frac{\sum_i c_i}{\sum_i d_i} = \frac{1}{\sum_i d_i} \left(\sum_{i : c_i \geq d_i} c_i + \sum_{i : c_i < d_i} c_i \right) \geq \frac{1}{\sum_i d_i} \cdot \sum_{i : c_i \geq d_i} d_i = 1 - \Pr_q[c_q < d_q],$$

and hence $\Pr_q[c_q < d_q] \geq \epsilon$. □

In particular, we apply the above claim to pairs (c_i, d_i) for $i \in U_v$. By the assumption that $\sum_{i \in U_v} c_i < \sum_{i \in U_v} (1-\epsilon) d_i$, we have that, by our choice of a random anchor q, $\Pr_q[q \in Z] = \Pr_q[c_q < d_q] > \epsilon$. Since we sample $h = \Omega(\frac{\log n}{\epsilon})$ anchors from U_v, at least one these anchors will belong to Z with high probability.

Now, for every fixed anchor $q \in Q_v \cap Z$, we prove that $(j_q, l_q) \in S_q$—remember that $c_q < d_q$ since $q \in Z$. We prove this by induction on the distance from the root. The base case is simple: when the node v is the root, this follows from the definition of $S_q = \{(j, l) : \mathrm{ed}(X_q, Y_{j,l}) < d_q\}$.

Now suppose v is not root, and all its ancestors are successful. Let r be v's parent. Since r is successful, there exists anchor $q' \in Q_r \cap Z$. By inductive hypothesis, $(j_{q'}, l_{q'}) \in S_{q'}$. By Lemma 2.3, the triples (q, j_q, l_q) and $(q', j_{q'}, l_{q'})$ are compatible. Hence $(j_q, l_q) \in \breve{S}_q$, and, since $\mathrm{ed}(X_q, Y_{j_q, l_q}) = c_q < d_q$, we have that $(j_q, l_q) \in S_q$, completing the inductive proof. □

By the above lemma, the entire tree has the following structure (whp): the "top" of the tree is composed of a set of successful nodes whose ancestors are also successful, and some of these nodes have unsuccessful children v, for which, by the above, we must have $\sum_{i \in U_v} c_i \geq (1-\epsilon) \sum_{i \in U_v} d_i$. Hence, we can partition the set I into set family $\{U_v\}_{v \in V}$ with the following property: each U_v, where $v \in V$ for some index set V, is either a successful singleton (leaf), or $\sum_{i \in U_v} c_i \geq \sum_{i \in U_v} (1-\epsilon) d_i$. Note that in the latter case, the edges (i, j_i, l_i), for $i \in U_v$, have cost at most $d_i + K_{\pi_i, (j_i, l_i)}$ (from the first phase). Hence, overall, the cost of edges (i, j_i, l_i) for $i \in I$ is at most:

$$\begin{aligned}
&\sum_{v \in V \text{ successful}} \sum_{i \in U_v} c_i + \sum_{v \in V \text{ not successful}} \sum_{i \in U_v} d_i + K_{\pi_i, (j_i, l_i)} \\
&\leq \sum_{v \in V \text{ successful}} \sum_{i \in U_v} c_i + \sum_{v \in V \text{ not successful}} \sum_{i \in U_v} 2d_i + c_i \\
&\leq \sum_{v \in V \text{ successful}} \sum_{i \in U_v} c_i + \sum_{v \in V \text{ not successful}} \sum_{i \in U_v} \frac{2c_i}{1-\epsilon} + c_i
\end{aligned}$$

$$\leq \sum_{i \in I} (3+4\epsilon) \cdot \mathrm{ed}(X_i, Y_{j_i,l_i}).$$

We've shown that the cost of the edges (i, j_i, l_i) is at most $3+4\epsilon$ times the real cost. Together with Lemma 2.3 this completes the proof of the correctness.

2.3 Analysis: run-time

Phase one of the algorithm takes time $O(k \cdot (p+b) \cdot t(w)) = O_\epsilon(kn/w \cdot \beta \cdot t(w))$.

In phase two, the main work is computing distances between each anchor q and its set $\breve{S}_q$. Hence, we need to show that the size of the sets $\breve{S}_q$, over all anchors q, is not too large overall. First we argue that each S_q is small. Indeed, the following claim suffices, recalling that $p = |J| \cdot |L|$:

Claim 2.6. *For any $i \in I$, there are at most $C \cdot p/k$ blocks $\tau \in J \times L$ such that $\mathrm{ed}(X_i, Y_\tau) < d_i$, whp for some large constant $C > 0$.*

Proof. Consider all the y-blocks $\tau \in J \times L$ in the increasing order of the distance from X_i. With high probability, the set of centers C includes a block of rank at most $O(p/k)$. In particular the closest center π_i has rank at most $O(p/k)$ and thus there are $O(p/k)$ blocks at a smaller distance. □

The anchors from the root perform $h \cdot p$ ed-calculations, for a total time of $hp \cdot t(n)$. Now fix a node v at level $j \geq 1$ (with the convention that the root is at level 0), and anchor $q \in Q_v$. Remember that $\breve{S}_q$ is composed of all (j,l) that are compatible with anything from $\cup_{q' \in Q_r} q' \times S_{q'}$, where r is the parent of v. Consider one of r's anchors $q' \in Q_r$, and let $(j', l') \in S_{q'}$. We want to upper-bound the number of pairs $(j,l) \in J \times L$ such that (q,j,l) is compatible with (q', j', l'), i.e., $|j - j'| \leq \frac{1}{\epsilon} \cdot |q - q'|$. Note that $|q - q'|$ is upper bounded by the diameter of $U_{q'}$ (as $q, q' \in U_{q'}$), which is $n \cdot 2^{-j+1}$. Hence, by compatibility property, we must have $|j - j'| \leq \frac{1}{\epsilon} \cdot n \cdot 2^{-j+1}$. Thus the number of compatible (j,l) is at most $2|L| \cdot \frac{1/\epsilon \cdot n \cdot 2^{-j+1}}{\Delta}$. The total number of ed-calls, for a fixed anchor q, is at most

$$|Q_r| \cdot \max_{q' \in Q_r} |S_{q'}| \cdot 2|L| \cdot \tfrac{1/\epsilon \cdot n \cdot 2^{-j+1}}{\Delta} \leq h \cdot Cp/k \cdot \tfrac{4|L|}{\epsilon^2} \tfrac{n\beta}{w} 2^{-j}$$

Over all nodes v and their anchors, the runtime becomes:

$$hp \cdot t(w) + \sum_{j=1}^{O(\log b)} 2^j \cdot h \cdot \left(h \cdot Cp/k \cdot \tfrac{4|L|}{\epsilon^2} \tfrac{n\beta}{w} 2^{-j} \right) \cdot t(w) \leq \tilde{O}(\tfrac{n\beta}{w} \cdot t(w)) + \tilde{O}(\tfrac{n\beta/w}{k} \cdot \tfrac{n\beta}{w} \cdot t(w)) \leq \tilde{O}(\tfrac{n^2\beta^2}{w^2 k} \cdot t(w)).$$

The third phase takes time $\tilde{O}(|I| \cdot |J| \cdot |L|) = \tilde{O}(n/w \cdot n/w \cdot \beta)$ and $w = (n/\beta)^{1.5}$:
Thus, overall runtime is, when choosing the optimizing $k = \sqrt{n\beta/w}$:

$$\tilde{O}_\epsilon((k\tfrac{n\beta}{w} + \tfrac{n^2\beta^2}{w^2 k}) \cdot t(w) + \tfrac{n^2}{w^2}\beta) = \tilde{O}_\epsilon(\tfrac{n^{1.5}\beta^{1.5}}{w^{1.5}} \cdot t(w) + \tfrac{n^2\beta}{w^2}). = \tilde{O}_\epsilon(n^{1.6}\beta^{1.4}).$$

2.4 Proof of Lemma on Reduction to block-matching

Proof of Lemma 2.3. To analyze some distance $\mathrm{ed}(x,y)$, we consider the optimal alignment $A : [n] \to [n] \cup \{\perp\}$ that certifies $\mathrm{ed}(x,y)$. In particular, we have that, $A(i) < A(j)$ for any $i < j \in [n] \setminus A^{-1}(\perp)$, and

$$\mathrm{ed}(x,y) = \#\left\{ i \in [n] \setminus A^{-1}(\perp) : x[i] \neq y[A(i)] \right\} + 2\left|A^{-1}(\perp)\right|.$$

Note that the starting and ending positions of the blocks in y are multiples of $\Delta = \max\{1, \epsilon/\beta \cdot w\}$; for convenience we define $\Delta' = \epsilon/\beta \cdot w$. We call them mini-blocks, with starting positions $S = \Delta \cdot [n/\Delta]$. For each $i \in [n/w]$ and corresponding block X_{iw}, we define $s_i \in S$ to be the first mini-block containing $A(z) \neq \perp$ for z in the block X_{iw}. Similarly define $t_i \in S$ to be the last such mini-block. If s_i, t_i do not exist, we define $s_i = t_{i-1} + 1$ and $t_i = t_{i-1}$ (where $t_{-1} = -1$ by convention). Note that we have that $t_i \leq s_{i+1}$ for all i.

The starting point is to match the block X_{iw} to string $y[s_i : t_i + \Delta]$, i.e., to set $j_i = s_i$ and $l_i = t_i - s_i + \Delta$. In that case the LHS in Eqn. (1) is upper-bounded by $\mathrm{ed}(x, y) + 2\Delta \cdot b$ (each block may introduce error of $\leq 2\Delta$ due to rounding). However, we also need to make sure that the intervals $[s_i : t_i + \Delta]$ satisfy the desired properties: they are disjoint, not too long, and are not too spread out. To accomplish this, we modify j_i, l_i in a few steps below, while controlling the incurred error.

We ensure disjointness as follows: for each $i \geq 1$, if $j_{i-1} + l_{i-1} = j_i + \Delta$, we set $j'_i = j_i + \Delta$ and $l'_i = l_i - \Delta$ (and $j'_i = j_i, l'_i = l_i$ otherwise as they are disjoint). Note that, for some blocks i, it may now be that $l'_i = 0$, in which case X_i just matches to an empty block. We now have that $j'_{i-1} + l'_{i-1} \leq j'_i$. The incurred error per block is $\leq \Delta'$.

Second, we ensure that block lengths are valid: in particular, that each block length $l'_i \in L$. We now set l''_i using l'_i as follows: if $l'_i < w$ we round it up to the nearest indez in L, and otherwise take the minimum between rounding down in L and $l''_i = w/\epsilon$. Let's analyze the incurred error. After changing from l'_i to l''_i, the LHS in Eqn. (1) increases by at most:

$$2 \cdot |l'_i - l''_i| = 2 \cdot |(l'_i - w) - (l''_i - w)| \leq 2\epsilon(1+\epsilon) \cdot \mathrm{ed}(X_{iw}, y[j'_i : j'_i + l'_i]).$$

where we've used the fact that $\mathrm{ed}(X_{iw}, y[j'_i : j'_i + l'_i]) \geq |l'_i - w|$. Hence, we get that:

$$\sum_i \mathrm{ed}(X_{iw}, y[j'_i : j'_i + l''_i]) + (j'_i - (j'_{i-1} + l''_{i-1})) \leq \mathrm{ed}(x, y) + O(\tfrac{n}{w} \cdot \Delta') + 2\epsilon(1+\epsilon) \cdot (\mathrm{ed}(x, y) + O(\tfrac{n}{w} \cdot \Delta'))$$

$$= (1 + O(\epsilon)) \,\mathrm{ed}(x, y) + O(\epsilon n/\beta).$$

We are left with the final property to ensure: that $|j_i - j_{i'}| \leq \frac{1}{\epsilon}|i - i'|$ for all $i' < i$. Note that it is enough to ensure this for $i' = i + 1$ (by triangle inequality). We ensure that by constructing j''_i from j'_i, iterating $i \in [n/w]$ in order. For current i, suppose $j'_i - j''_{i-1} > w/\epsilon$. Then we simply set $j''_i = j''_{i-1} + w/\epsilon$ (note that $j''_i - j''_{i-1} = w/\epsilon \geq l''_{i-1}$, so keeping the same lengths is ok), and leaving $j''_i = j'_i$ otherwise. Note that the LHS can increase only by at most w as some characters from X_{iw} may lose their matches, while $\sum_i j''_i - (j''_{i-1} + l''_{i-1})$ remains the same overall. To account for this increase in cost, we "charge" this cost to

$$(j''_i - (j''_{i-1} + l''_{i-1}) + \mathrm{ed}(X_{(i-1)w}, Y_{j''_{i-1}, l''_{i-1}}) = w/\epsilon + \Big(\mathrm{ed}(X_{(i-1)w}, Y_{j''_{i-1}, l''_{i-1}}) - l''_{i-1}\Big) \geq w/\epsilon - 2w \geq w/2\epsilon.$$

Since $\sum_i j''_i - (j''_{i-1} + l''_{i-1})$ remains the same overall, the extra cost is only at most factor 2ϵ of the total cost when using indeces j'_i, l''_i. Hence the the changes here increases the total cost by a factor of at most $1 + 2\epsilon$.

The final output are the indeces (j''_i, l''_i). □

3 Improving runtime with a text searching primitive

The main algorithm from the previous section uses the subroutine of computing exact edit distance between two strings of length w. Since many of these distances are computed for overlapping y-blocks, we can actually exploit a slightly stronger primitive, to obtain a slightly faster runtime. The primitive is *text searching*: given a pattern $p \in \Sigma^w$, another text $T \in \Sigma^n$, and a length $l = O(w)$, compute distance from p to each substring of T of length l. When $n = O(w)$, we call its runtime $t(w)$. Note that, by splitting a longer text, of length n, into n/l overlapping blocks of length $2l$, and running text search on each, we get that $t(n, w) = O(n/l \cdot t(w))$.

Theorem 3.1. *For any $\epsilon > 0, \beta > 1$, we can estimate edit distance between two strings of length n in time $\tilde{O}_\epsilon(n^{1.6}\beta^{0.6})$, achieving $(3+\epsilon)$-factor and $\pm n/\beta$ additive approximation. When further combined with the exact $O(n + (n/\beta)^2)$-time algorithm [Ukk85, Mye86], we obtain $O_\epsilon(n^{1.693})$ time.*

We now discuss two aspects of this modification: 1) how to implement the new primivite, and 2) how to use it to improve the main theorem.

Implementing the text searching primitive. In [Sel80], the author modified the standard dynamic programming solution to solve the following problem: for every end position j in T, find a substring of T at smallest edit distance from p (together with the distance value). The standard dynamic programming is modified by adding zero-cost edges from the start $(0, 0)$ to each $(0, s)$, $s \in [w]$ and then computing single-source shortest path to all (w, j). Note that, one can similarly solve the following primitive: given a starting set $S \subset [w]$, compute $\min_{s \in S} \mathrm{ed}(p, T[s : j])$ for each $j \in [n]$. This is done by adding edges only to $(0, s)$ for $s \in S$. Call the latter problem $TSS(p, T, l, S)$, and note that it can output a vector $d_S[1 : n]$ of distances to (w, j)'s, together with a vector $r_S[1 : n]$ where $r_S[j]$ is the index i such that the shortest path to (w, j) passes through $(0, i)$.

We now design an algorithm for our suggested primitive, achieving $1 + 8\epsilon$ approximation. First of all, we can assume that l is within a factor of $1/\epsilon$ of w as otherwise $\max\{l, w\}$ is a $(1 + \epsilon)$-factor approximation. Now, for each "scale" power of two $c \in \{1, 2, 4, \ldots w/\epsilon\}$, we perform the call $TSS(p, T, l, S_c)$ where $S_c = \{\epsilon ck \mid k \in \mathbb{N}\} \cap [w]$. Note that each of them yields an estimate $E_c(j)$ for $\mathrm{ed}(p, T[j : j + l])$, namely $d_{S_c}[j + l - 1] + \left|j - r_{S_{c,i}}[j + l - 1]\right|$. It's immediate to check that the estimate is an upper bound: it's the sum of matching a suffix of $T[j : j + l]$ to p together with the cost of inserting the missed prefix.

The final estimate for $\mathrm{ed}(p, T[j : j + l])$ for $j \le n - l$ is the minimum of $E_c(j)$ over all c, which we claim is a $1 + O(\epsilon)$ approximation. Indeed, let $v = \mathrm{ed}(p, T[j : j + l])$. Fix c to be the smallest power of 2 larger than $3v$, and k be the integer such that $s = \epsilon ck$ is closest to j. Note that $|s - j| \le \epsilon c/2 < 3\epsilon v$. Since $r_{S_{c,i}}[j] = s$ is an option for the shortest path, $E_c(j) \le \mathrm{ed}(p, T[j : j + l]) + 3\epsilon v + 3\epsilon v \le (1 + 6\epsilon) \cdot \mathrm{ed}(p, T[j : j + l])$.

The runtime is $\tilde{O}(nw)$.

Improving main algorithm with the text searching primitive. The modification to the algorithm from Section 2 is immediate: whenever we need to compute edit distance from some block p (from x or y) to many blocks in y, we do so using the new primitive. Specifically, denoting by $\mathcal{A}(p, T, l)$ our new primitive:

- The quantities $K_{\pi,\tau}$ are computed using $k \cdot |L|$ $\mathcal{A}$ calls (one per y-center and length $l \in L$).

- For each top-level anchor $q \in I$, we compute the edit distance from X_q to each block Y_π for $\pi \in J \times L$ using one $\mathcal{A}$ call for each $l \in L$.
- Finally, we use $\mathcal{A}$ to compute the distance from anchors X_q to Y_π for $\pi \in \breve{S}_q$, where we need to be a bit careful as $\breve{S}_q$ is not contiguous. For fixed length $l \in L$, we partition J into n/l overlapping blocks each of size $2\frac{|J|}{l}$, termed $H_j^l = [jl : jl + 2l]$ for $j \in [n/l]$. For each $l \in L$, and $j \in [n/l]$, we run $\mathcal{A}(X_q, y[H_j^l], l)$ if $[jl : jl + l] \cap \breve{S}_q \neq \emptyset$. Note that this covers all relevant substrings encoded by $\breve{S}_q$.

We now analyze the new runtime, recalling that, for $t(n, w)$ denoting the runtime of $\mathcal{A}$ on pattern of length w and text of length n, we have that $t(n, w) = O(t/w \cdot t(w))$ where $t(w) = t(w/\epsilon, w)$.

- For phase 1, the runtime is now $\tilde{O}(k \cdot t(n, w) + bk \cdot t(w)) = \tilde{O}(kn/w \cdot t(w))$.
- For phase 2, first note that the anchors from the root perform h calls to $\mathcal{A}$, for a total time of $h \cdot t(n, w)$. For computing distance to $\breve{S}_q$, we note that, for fixed $l \in L$, all compatible (j, l) are covered by at most $2\frac{1/\epsilon \cdot n \cdot 2^{-j+1}}{l}$ blocks H, yielding an upper bound on the number of $\mathcal{A}$ calls of $2\frac{1/\epsilon \cdot n \cdot 2^{-j+1}}{\epsilon w} = \frac{4}{\epsilon^2}\frac{n}{w} \cdot 2^{-j}$, each with $t(w)$ time. The total number of $\mathcal{A}$ calls, for a fixed anchor q, is at most

$$|Q_r| \cdot \max_{q' \in Q_r} |S_{q'}| \cdot \tfrac{4}{\epsilon^2}\tfrac{n}{w}2^{-j} \cdot |L| \leq h \cdot Cp/k \cdot \tfrac{4}{\epsilon^2}\tfrac{n}{w}2^{-j} \cdot |L|.$$

  Over all nodes v and their anchors, the runtime becomes:

$$h \cdot t(n, w) + \sum_{j=1}^{O(\log b)} 2^j \cdot h \cdot \big(h \cdot Cp/k \cdot \tfrac{4}{\epsilon^2}\tfrac{n}{w}2^{-j} \cdot |L|\big) \cdot O(t(w)) \leq \tilde{O}(t(n, w)) + \tilde{O}(\tfrac{n/w \cdot \beta/\epsilon}{k} \cdot \tfrac{n}{w} \cdot t(w)) \leq \tilde{O}(\tfrac{n^2\beta}{w^2 k} \cdot t(w)).$$

- Thus, overall runtime is, when choosing the optimizing $k = \sqrt{n\beta/w}$ and $w = (n\beta)^{0.2}$:

$$\tilde{O}_\epsilon((\tfrac{kn}{w} + \tfrac{n^2\beta}{w^2 k}) \cdot t(w) + \tfrac{n^2}{w^2}\beta) = \tilde{O}_\epsilon(\tfrac{n\sqrt{n\beta}}{w^{1.5}} \cdot w^2 + \tfrac{n^2\beta}{w^2}) = \tilde{O}(n^{1.6}\beta^{0.6}).$$

References

[AHWW16] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 375–388. ACM, 2016.

[AKO10] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2010. Full version at `http://arxiv.org/abs/1005.4033`.

[AO12] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM J. Comput. (SICOMP)*, 41(6):1635–1648, 2012. Previously in STOC'09.

[AS20] Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2020.

[BEG+18] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1170–1189. SIAM, 2018.

[BEK+03] Tuğkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 316–324, 2003.

[BES06] Tuğkan Batu, Funda Ergün, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 792–801, 2006.

[BI15] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Symposium on Theory of Computing (STOC)*, 2015.

[BJKK04] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 550–559, 2004.

[BR19] Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. *arXiv preprint arXiv:1904.05390*, 2019.

[CDG+18] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 979–990. IEEE, 2018.

[GKS19] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1101–1120. IEEE, 2019.

[GRS20] Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In *Proceedings of the Symposium on Theory of Computing (STOC)*, 2020.

[KS19] Michal Koucký and Michael E Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. *arXiv preprint arXiv:1904.05459*, 2019.

[KS20] Tomasz Kociumaka and Barna Saha. Sublinear-time algorithms for computing & embedding gap edit distance. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2020.

[MP80] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[Mye86] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[RSSS19] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for lcs and lis with truly improved running times. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1121–1145. IEEE, 2019.

[Sel80] Peter H Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms*, 1(4):359–373, 1980.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168 – 173, 1974.