Edit Distance in Near-Linear Time: it's a Constant Factor

Alexandr Andoni
Columbia University

Negev Shekel Nosatzki
Columbia University

2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS) | 978-1-7281-9621-3/20/$31.00 ©2020 IEEE | DOI: 10.1109/FOCS46700.2020.00096

***Abstract*—We present an algorithm for approximating the edit distance between two strings of length n in time $n^{1+\epsilon}$, for any $\epsilon > 0$, up to a constant factor. Our result completes a research direction set forth in the recent breakthrough paper [1], which showed the first constant-factor approximation algorithm with a (strongly) sub-quadratic running time. The recent results [2], [3] have shown near-linear complexity only under the restriction that the edit distance is close to maximal (equivalently, there is a near-linear additive approximation). In contrast, our algorithm obtains a constant-factor approximation in near-linear running time for any input strings.**

***Keywords*-edit distance; sublinear algorithms; fine-grained complexity**

I. Introduction

Edit distance is a classic distance measure between sequences that takes into account the (mis)alignment of strings. Formally, edit distance between two strings of length n over some alphabet Σ is the number of insertions/deletions/substitutions of characters to transform one string into the other. Being of key importance in several fields, such as computational biology and signal processing, computational problems involving the edit distance were studied extensively.

Computing edit distance is also a classic dynamic programming problem, with a quadratic run-time solution. It has proven to be a poster challenge in a central theme in TCS: improving the run-time from polynomial towards close(r) to linear. Despite significant research attempts over many decades, little progress was obtained, with a $O(n^2/\log^2 n)$ run-time algorithm [4] remaining the fastest one known to date. See also the surveys of [5] and [6]. With the emergence of the fine-grained complexity field, researchers crystallized the reason why beating quadratic-time is hard by connecting it to the Strong Exponential Time Hypothesis (SETH) [7] (and even more plausible conjectures [8]).

Even before the above hardness results, researchers started considering faster algorithms that approximate edit distance. A linear-time $\sqrt{n}$-factor approximation follows immediately from the exact algorithm of [9], [10], [11], which runs in time $O(n + d^2)$, where d is the edit distance between the input strings. Subsequent research improved the approximation factor, first to $n^{3/7}$ [12], then to $n^{1/3+o(1)}$ [13], and to $2^{\tilde{O}(\sqrt{\log n})}$ [14] (based on the ℓ_1 embedding of [15]). In the regime of $O(n^{1+\epsilon})$-time algorithms, the best approximation is $(\log n)^{O(1/\epsilon)}$ [16]. Predating some of this work was the sublinear-time algorithm of [17] achieving n^{ϵ} approximation when d is large.

In a recent breakthrough, [1] showed that one can obtain constant-factor approximation in $O(n^{12/7})$ time. Subsequent developments [2], [3] give $O(n^{1+\epsilon})$-time algorithms for computing edit distance up to an *additive* $n^{1-g(\epsilon)}$ *term* and $f(1/\epsilon)$-factor approximation, for some non-decreasing functions f, g, and any $\epsilon > 0$.

Our main result is a $n^{1+\epsilon}$ algorithm for computing the edit distance up to a constant approximation.

Theorem I.1. *For any $\epsilon > 0$, $n \geq 1$, alphabet Σ, and two strings $x, y \in \Sigma^n$, there's an algorithm to approximate edit distance between x, y in $O(n^{1+\epsilon})$ time up to $f(1/\epsilon)$-factor approximation, where $f(1/\epsilon)$ depends solely on ϵ.*

While we do not derive the function $f(1/\epsilon)$ explicitly, we note that it is doubly exponential in $1/\epsilon$. We present a technical overview of our approach in Section III, after setting up our notations in Section II. The top-level algorithm, its main guarantees, and how they imply the above theorem are in Section IV. The proof of the main guarantees, where the most of the work is happening, appears in the full version of this extended abstract, at `https://arxiv.org/abs/2005.07678`.

A. Related work

A quantum algorithm for edit distance was introduced in [18]. Some of the basic elements of the algorithmic approach are related to [1] (and the algorithm in this paper). Another recent related paper is [19], who obtain $3+\epsilon$ approximation in $\tilde{O}(n^{1.6})$ time; independently, the first author obtained a slightly worst time for the same approximation [20]. Similarly, independently, [21] and [20] extended the constant-factor edit distance algorithm from [1] to solve the text searching problem.

A sublinear time algorithm was also recently developed in [22]; see also earlier [12], and the aforementioned [17]. Another related line of work has been on computing edit distance for the semi-random models of input [23], [24]. Parallel (MPC) algorithms were developed in [18], [25].

Progress on edit distance algorithms also inspired the first non-trivial algorithms for approximating the longest common subsequence (LCS) [26], [27], [28]. In fact, [28] show that a $O(1)$-factor approximation to edit distance yields

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00096

a $2 - \Omega(1)$ factor approximation to LCS over a binary alphabet in the same time.

B. Acknowledgements

Research supported in part by NSF grants (CCF-1617955 and CCF-1740833), and Simons Foundation (#491119).

II. Preliminaries: Setup and Notations

Fix a pair of strings $(x, y) \in \Sigma^n \times \Sigma^n$ for which we care to estimate the edit distance. We define $\texttt{ed}_n(x, y)$ as half the number of insertions/deletions to transform one string into the other. Note that this is a factor-2 approximation to the standard edit distance. When length n is clear from the context, we omit the subscript.

S_w is the set of powers of 2 up to w: namely, $S_w = \{0, 1, 2, 4, 8 \dots w\}$. $[n]$ denotes set $\{1, 2, 3, \dots n\}$, throughout the paper, except where stated explicitly.

When describing intuitive parts, we sometimes use $O^*(f(n))$ to denote $O(f(n) \cdot n^{O(\epsilon)}$ (where ϵ is the small constant from the algorithm).

A. Intervals

An interval is a substring $x[i : j] \triangleq x_i x_{i+1} \dots x_{j-1}$, for $i, j \in [n]$, where $i \le j$ (i.e., starting at i and ending at $j-1$, of length $j - i$).

For $i \in [n]$, let $X_{i,w}$ ($Y_{i,w}$) denote the interval of x (y) of length w starting at position i. Let $\mathcal{X}_w, \mathcal{Y}_w$ the set of all such $X_{i,w}$ and $Y_{j,w}$ strings respectively. We use $\mathcal{I}_w = \mathcal{X}_w \cup \mathcal{Y}_w$ to denote all x and y axis intervals. When clear from context, we drop subscript w.

By convention, if $i \notin [1, n-w]$, we pad X_i/Y_i with a default character, say, \$. Also $Y_{\perp,w}$ is a string of unique characters. In particular, for various distance functions $\tau_w(\cdot, \cdot)$ on two length-w strings, $\tau(X_{i,w}, Y_{\perp,w})$ is the max possible distance; e.g., $\texttt{ed}_w(X_{i,w}, Y_{\perp,w}) = w$.

Usually, by $I \in \mathcal{I}_w$ we refer not only to the corresponding substring but also to the "meta-information", in particular the string it came from, start position, and length (e.g., for $I = X_{i,w}$, the meta-information is x, i, w). This difference will be clear from context or stated explicitly.

In particular, the notation $I + j$, for an interval I and integer j, represents the interval j positions to the right; e.g., if $I = X_{i,w}$, then $I + j = X_{i+j,w}$.

Alignments. An alignment between x and y which is a function $\pi : [n] \to [n] \cup \{\perp\}$, which is injective and strictly monotone on $\pi^{-1}([n])$. The set of all such alignments is called Π. Note that $\texttt{ed}(x, y) = \min_{\pi \in \Pi} \sum_{i \in [n]} \texttt{ed}_1(x_i, y_{\pi(i)})$ (recall that, by convention, $\texttt{ed}_1(c, y_\perp) = 1$ for all $c \in \Sigma$).

It is convenient for us to think of π as function from $\mathcal{I} \to \mathcal{I}$, via the following extension. For a given input alignment $\pi : \mathcal{X} \to \mathcal{Y} \cup \{\perp\}$, its extension $\widehat{\pi} : \mathcal{I} \to \mathcal{I} \cup \{\perp\}$ is:

$$\widehat{\pi}[I] = \begin{cases} \pi[I] & I \in \mathcal{X} \\ \pi^{-1}[I] & I \in \mathcal{Y} \end{cases},$$

where $\widehat{\pi}[X_{i,w}]$ means $Y_{\pi(i),w}$, and $\widehat{\pi}[Y_{j,w}]$ means $X_{\pi^{-1}(j),w}$, with $\pi^{-1}(j) = \perp$ if there's no i with $\pi(i) = j$. Throughout this paper, we overload notation to use π for the extension $\widehat{\pi}$ as well. We also define $\overleftarrow{\pi}(i)$ as the minimum $\pi(j)$, $j \ge i$, which is defined ($\neq \perp$).

Finally, we also define $\pi(i) \triangleq i$ when $i < 1$ and $i > n$ for convenience.

B. Interval distances

Our algorithms will use distances/metrics over intervals in $\mathcal{I}_w$. One important instance is the *alignment distance*, denoted $\texttt{ad}_w(\cdot, \cdot)$. At a high level, $\texttt{ad}_w(\cdot, \cdot)$ is a distance metric that approximates edit distance on length-w intervals. We discuss $\texttt{ad}(\cdot, \cdot)$ metric also in Section IV.

Definition II.1 (Neighborhood)**.** *Fix $c \ge 0$ and $I \in \mathcal{I}_w$. The* c-neighborhood *of I is the set $\mathcal{N}_c(I) = \{J \in \mathcal{I}_w \mid \texttt{ad}(I, J) \le c\}$, i.e. all x and y intervals which are c-close to I in terms of their alignment distance.*

Definition II.2 (Ball of intervals)**.** *A* ball of intervals *is a set of consecutive intervals in either $\mathcal{X}_w$ or $\mathcal{Y}_w$ (i.e., it's a ball in the metric where distance between X_i and X_j is $|i - j|$). The* smallest enclosing ball *of a set $\mathcal{S}$ is the minimal ball $\mathcal{B} \supseteq \mathcal{S}$.*

C. Operations on sets and the $*$ notation

By convention, applying numerical functions to a set refers to the sum over all set items; e.g., $f(S) = \sum_{i \in S} f(i)$. When applying set operators on other sets, we use the union; e.g., $\pi(\mathcal{S}) = \cup_{I \in \mathcal{S}} \pi(I)$ and $\mathcal{N}_c(\mathcal{S}) = \cup_{I \in \mathcal{S}} \mathcal{N}_c(I)$. Any exception to the above will be clearly specified.

We also use the notation $*$ as argument of a function, by which we mean a vector of all possible entries. E.g., $f(*)$ is a vector of $f(i)$ for i ranging over the domain of f (usually clear from the context). Similarly, $f(*_{\mathcal{R}})$ means a vector of $f(i)$ for i satisfying property $\mathcal{R}$.

III. Technical Overview

A. Prior work and main obstacles

As our natural starting point is the breakthrough $O(n^{12/7})$-time algorithm of [1], we first describe their core ideas as well as the challenges to obtaining a near-linear time algorithm. In particular, we highlight two of their enabling ideas. At a basic level, their algorithm computes edit distance $\texttt{ed}_n(x, y)$ by computing $\texttt{ed}_w$ between various length-w *intervals* (substrings) of x, y recursively, and then uses edit-distance-like dynamic programming on intervals to put them back together. The main algorithmic thrust is to reduce the number of recursive $\texttt{ed}_w$ computations: e.g., if the intervals are of length w, and we only consider non-overlapping intervals, there are still $n/w \times n/w$ calls to

$\texttt{ed}_w$, each taking at best $\Omega(w)$ time. Hence, [1] employ two ideas to do this efficiently: 1) use the triangle inequality to deduce distance between pairs of intervals for which we do not directly estimate $\texttt{ed}_w$, 2) two nearby x-intervals (e.g., consecutive) are likely to be matched into two nearby y-intervals (also consecutive) under the optimal edit distance alignment π. Indeed, these ideas are enough to reduce the number of recursive calls from $(n/w)^2$ to $\approx (n/w)^{1.5}$.

One big challenge in the above is that, in general, one has to consider all, overlapping intervals from x, y, of which there are n — since, in an optimal $\texttt{ed}_n$ alignment, an x-interval might have to match to a y-interval whose start position is far from an integer multiple of w. An alternative perspective is that if one considers only a restricted set of interval start positions, say every $s \leq w$ positions in y, then one obtains an extra *additive* error of about $s \cdot n/w$ from the "rounding" of start positions in y. That's the reason that a bound of $(n/w)^{1.5}$ recursive calls did not transform into $n^{1.5}$ runtime in [1]: to compute edit distance when $\texttt{ed} < n^{1-\Omega(1)}$, they employ a standard (exact) $\tilde{O}(n + \texttt{ed}^2(x, y))$ algorithm [9], [10].

While recent improvements by [2], [3] showed how to reduce the number of recursive calls to $\approx n/w$, some fundamental obstacles remained. The linear number of recursive calls was leveraged to obtain near-linear time but with an additive approximation only: when $\texttt{ed}(x, y) \geq n^{1-\delta}$, the overall runtime is $n^{1+f(\delta)}$ for some increasing function f.

In particular, in addition to the aforementioned challenge, a new challenge arose: to be able to reduce to near-linear number of recursive $\texttt{ed}_w$ calls, the algorithms from [2], [3] might miss a large fraction of "correct" matches. In particular this fraction is $\approx n^{-\delta}$, which results in an additive error of $\approx n^{1-\delta}$. To put this into perspective, for $w = \sqrt{n}$, if we allow an additive error $n^{1-\delta}$, then it suffices to analyze $b = n^{0.5+O(\delta)}$ intervals (which barely overlap) and misclassify $b \cdot n^{-\delta}$ of them.

As a running example illustrating the challenges, consider an instance where $\Delta = n^{-0.01}$ fraction of intervals in $\mathcal{X}_w$ are "sparse" — have a single cheap match (under π) in $\mathcal{Y}_w$ — and the rest of the intervals are *dense* (they have large cheap ed-neighborhoods). Assume further that such sparse intervals are spread around in multiple *sparse sections*. Note that if we can afford large additive errors, we can simply *ignore* all these sparse intervals (certifying them at max cost w) and output the distance based on the dense intervals only, with at most $n\Delta = n^{0.99}$ additive approximation. To avoid this, one must first identify some sparse intervals (since the dense intervals do not provide sufficient information about the sparse sections). Even if we manage to find some of the sparse intervals efficiently, we still need to apply knowledge of the location of such intervals to deduce information on other intervals which might be in completely different areas in the string. We will return to this example later.

Below we describe the high-level approach to our algorithm, including how we overcome these obstacles. We note that, except for the above two core ideas from [1], we depart from the general approach undertaken in [1], [2], [3]. We also do not rely on previous results for any distance regime.

B. Our high-level approach

While there are many ideas going in overcoming the above challenges, one common theme is *averaging over the local proximity of intervals*. In particular, the algorithm proceeds by, and analyzes over, "average characteristics" of various intervals of x, y, in a "smooth" way. For example decisions for a fixed interval $I \in \mathcal{I}_w$, such as whether something is close, or something is matched, are done by considering the statistics collected on nearby intervals (to the left/right of I in the corresponding string). While we expand on our technical ideas below, this is the guiding principle to keep in mind.

Addressing the first challenge, we consider intervals (of fixed length w) at all n starting positions, i.e., the entire set $\mathcal{I}_w$. Note that recursion becomes prohibitive: we can't perform even n edit distance evaluations each taking $\Omega(w)$ time (w is set to be $\approx n^{1-\epsilon}$). Instead, our top-level algorithm iterates bottom–up over all interval lengths $w = \gamma, \gamma^2, \ldots n$, where $\gamma = n^{\epsilon}$, and for each w computes a good-enough approximation to *the entire metric* $(\mathcal{I}_w, \texttt{ed}_w)$. Recall that $\mathcal{I}_w = \mathcal{X}_w \cup \mathcal{Y}_w$ consists of all w-length intervals (substrings); i.e., $|\mathcal{I}_w| = 2n$. The metric will be accessible via a distance oracle (fast data structure), with n^{ϵ} query time, and will $O(1)$-factor approximate most of the "relevant" matches of x-intervals to y-intervals. This approximating metric distance is called $\mathfrak{D}_w(\cdot, \cdot)$. In particular, if π is an optimal ed-alignment of x to y, then $\mathfrak{D}_w(X_{i,w}, Y_{\pi[i],w})$ will approximate $\texttt{ed}(x[i : i + w - 1], y[\pi[i] : \pi[i] + w - 1])$, on average, for all but $\approx \texttt{ed}(x, y)$ indeces $i \in [n]$.

In each iteration, we build $\mathfrak{D}_w$ using $\mathfrak{D}_{w'}$, where $w' = w/\gamma$. Conceptually we do so in two phases. First, we build another metric, $(\mathcal{I}_w, \texttt{ad}_w)$, accessible via a fast distance oracle, that uses $\gamma^{O(1)} = n^{O(\epsilon)}$ time and $\mathfrak{D}_{w'}$ oracle calls. Intuitively, $\texttt{ad}_w$ will similarly approximate $\texttt{ed}_w$. Second, equipped with a fast oracle for $\texttt{ad}_w$ (itself using $\mathfrak{D}_{w'}$), we build an "efficient representation" for the entire metric $(\mathcal{I}_w, \texttt{ad}_w)$, while using only $n^{1+O(\epsilon)}$ calls to $\texttt{ad}_w$ oracle. Naturally, this "efficient representation" will not be able to capture the entire $\texttt{ad}_w$ metric (whose description complexity could in general be $\gtrsim n^2$), but it will capture just enough to preserve the edit distance between x and y. Then we build an efficient distance oracle for this efficient representation, which will yield the desired metric $\mathfrak{D}_w$. Note that the final answer is computed by (essentially) querying $\mathfrak{D}_n(X_{1,n}, Y_{1,n})$.

In particular, the "efficient representation" of $\texttt{ad}_w$ is a weighted graph G_w with vertex set $\mathcal{I}_w$ and $n^{1+\epsilon}$ edges, such that the shortest path between $I, J \in \mathcal{I}_w$ approximates $\texttt{ad}_w(I, J)$. In particular, the shortest path distance is

non-contracting, and non-expanding for interval pairs that "matter", i.e., which are part of the optimal alignment π corresponding to $\texttt{ed}(x, y)$. An edge (I, J) of the graph G_w will always correspond to an explicit call to $\texttt{ad}_w(I, J)$; and the main question in constructing G_w is deciding *which* $n^{1+\epsilon}$ pairs to compute ad for.

Once we have the graph G_w, we build a fast distance oracle data structure on it to obtain the metric $\mathfrak{D}_w$. In particular, our fast distance oracle is merely an embedding of the shortest path metric on G_w into ℓ_∞^d, where $d = |\mathcal{I}|^\epsilon$, incurring an approximation of $O(1/\epsilon)$, via [29]. We note that we cannot use a more "common" distance oracle, such as, e.g., [30], [31], because they do not guarantee that the resulting output is actually a metric, and in particular, that it satisfies the triangle inequality, which is crucial for us (as mentioned above). We remark that this step is somewhat reminiscent of the approach from [14], who similarly build an efficient representation for the metric $(\mathcal{I}_w, \texttt{ed}_w)$ using metric embeddings. However, the similarity ends here: first [14] used Bourgain's embedding into ℓ_1, which incurs $\Theta(\log n)$ distortion, and second, more importantly, the construction of G_w was altogether different (incurring a much higher approximation).

Computing the graph G_w itself is the most algorithmically novel part of our approach, and is termed *Interval Matching Algorithm*, as it corresponds to matching intervals according to their $\texttt{ad}_w$ distance. This algorithmic part should be thought of as the analogue of the algorithm deciding for which pairs of intervals to (recursively) estimate the edit distance in [1].

We sketch the Interval Matching Algorithm next in this technical overview. We also sketch how to compute the $\texttt{ad}_w$ distance in $n^{O(\epsilon)}$ time, which presents its own challenges.

C. Interval matching algorithm

The main task here is to efficiently compute graph G_w so that for any pair $(I, \pi[I])$, where π is the optimal ed alignment, the shortest path between I and $\pi[I]$ in G_w is $O(\texttt{ad}_w(I, \pi[I]))$, on average, for all but $\lesssim \texttt{ed}(x, y)$ intervals. To generate G_w, we iterate over all magnitudes of costs $c \in S_w$, and for each such cost, we generate (sub-)graph $G_{w,c}$, with edges of weight $\Theta(c)$. In particular, for nearly all pairs $(I, \pi[I])$ at a distance $\texttt{ad}(I, \pi[I]) \le c$, we eventually generate a 1- or 2-hop path for it in $G_{w,c}$. The union[1] of such graphs $G_{w,c}$ yields the final graph G_w. Below we focus on a single scale graph $G_{w,c}$, which is supposed to capture all pairs $(I, \pi[I])$ where $\texttt{ad}(I, \pi[I]) \le c$. We refer to such a pair as a *π-matchable pair* $(I, \pi[I])$.

At its core, our algorithm can be thought of as a *partitioning* algorithm, where we partition $\mathcal{I}_w$ into sets of intervals, such that for nearly all π-matchable pairs $(I, \pi[I])$, both intervals belong to the same set. We start from large (coarse) partition and iteratively refine it into a smaller partition, keeping π-matchable pairs $I, \pi[I]$ together.

In particular, the matching algorithm proceeds in $\approx 1/\epsilon$ steps. In each step t, for $\lambda = n^\epsilon$, we generate λ^t parts. To construct a part, we sample a random interval A, termed *anchor*, and estimate $\texttt{ad}_w(A, I)$ for all other intervals I in its part, generating a *cluster* of intervals at distance $O(c)$ from the anchor. The main desideratum is that the two intervals from a π-matchable pair $I, \pi[I]$ are either both close to A or both far from A, and hence always remain together (this is related to the triangle inequality idea from [1]). However, this cannot be guaranteed, and ensuring this desideratum is a major challenge for us, which we will address later. For now, in order to build intuition, we first make the following assumption that ensures this desideratum:

Perfect Neighborhood Assumption (PNA): any two intervals are at distance either $\le c$ or $\omega(c)$; hence $\mathcal{N}_{O(c)}(I) = \mathcal{N}_c(I)$.

From a cluster, we construct one part (set) by taking the clustered intervals together with their *local extensions*: intervals around the clustered intervals (i.e., to the left/right of the clustered ones). The parameters are set up such that the resulting part has size $\lesssim n/\lambda^t$. As the partition granularity decreases with step t, we can afford to use more anchors: e.g., at step t, we start with λ^{t-1} partitions, each of size about n/λ^{t-1}, and hence, for each of λ^t anchors, we need to estimate ad distance to n/λ^{t-1} intervals (in its part), for an overall of $n\lambda$ distance computations.

A direct implementation of partitions as above however runs in various issues, yielding additive errors. In particular, it only guarantees to "correctly partition a π-matchable pair with some probability", instead of the needed "with some probability, all except a few π-matchable pairs are partitioned correctly" (aka, "for each" vs "for all" guarantee). For the latter goal, bounding the "except a few" so that it's only a $O(1)$-factor approximation, we use the notion of *corruption*, defined later.

Colorings. To describe a partition, we use the slightly generalized concept of a *coloring*: a coloring κ is a mapping from each interval $I \in \mathcal{I}$ to a *distribution of colors* in a color-set ν, where a fixed color should be thought of as a part. We denote the mapping by $\mu_\kappa : \mathcal{I} \times \nu \to [0, 1]$. For each interval I, we require $\|\mu_\kappa(I, *)\|_1 = 1$, i.e., we think of the interval I as being split into fractions each assigned to a part: fraction $\mu_\kappa(I, \chi) > 0$ is assigned to color χ. While under the "perfect neighborhood assumption", partitions are sufficient (i.e., $\mu_\kappa \in \{0, 1\}$), fractional colorings will be crucial for removing the assumption.

All but two colors $\chi \in \nu$ correspond to a part constructed from a fixed anchor (i.e., its cluster of intervals together with the local extension). There are also two special "colors": 1) the $\mathbf{u}$-color ("uncolored") consists of intervals which so

[1]When there are multiple (I, J) edges from different c's, we naturally take the minimum-weight edge—i.e., the smallest distance certificate.

far has failed to be captured in a part and remains "tbd" (intervals with certain "sparsity" properties, to be discussed later), and 2) the color $\perp$ that corresponds to the *already-matched* intervals, i.e., intervals for which we've already added a short path to their π-match in the graph $G_{w,c}$ (typically "dense" intervals that have already "converged"). Overall $\nu = [\lambda^t] \cup \{\mathbf{u}, \perp\}$.

Coloring construction via potentials. To construct a new, more refined step-t coloring (from the step-$(t-1)$ coloring), we design a mechanism for assigning *potential scores* to clustered intervals. Using these potential scores, we assign colors[2] to other nearby intervals in thier proximity, as suggested above. The main intuition is that a π-matchable pair $I, \pi[I]$ typically has a large set of other π-matchable $J, \pi[J]$ in their respective proximities (i.e., to the left/right).

How large of a "proximity" a cluster can color depends on the size of the ad-neighborhood of A (and hence of the clustered intervals, by the perfect neighborhood assumption). To quantify this, we introduce the notion of *density* of an interval I of color χ, termed $\mathrm{d}(I, \chi)$: the measure μ of its ad-neighborhood $\mathcal{N}_c(I)$ that share the color χ. If such I (and hence it's aligned $\pi[I]$) is "dense" (large $\mathcal{N}_c$), then we have a higher probability to cluster such a pair to an anchor; but we can only afford a small extension for each one (i.e., each clustered interval is used to color few other nearby intervals). In contrast, "sparse" matches will be clustered with a small probability, but can be used to generate large extensions in their proximity.

In particular, to compute the new step-t coloring, we color the intervals gradually in levels, indexed by $l = 0, 1, \ldots, 1/\epsilon$, each level taking care of a density regime as above. In each level, we define potentials ϕ and φ. First, for each anchor A, we allocate potential $\phi(I) \approx \frac{n}{d \cdot \lambda^t}$ to each clustered interval I (i.e., I at distance $O(c)$ from A in the same "part"[3] as A) of density d. Next, we define a derivative potential φ by splitting the allocated potential $\phi(I)$ across the $\mathbf{u}$-colored intervals in a proximity ball of radius $\zeta = \zeta(l) \approx n^{\epsilon l}$ around each clustered interval I. At the end of each level, we transform potential φ into new colors in μ_κ, decreasing the respective $\mathbf{u}$-color (to be discussed later). Overall, the following is a high-level diagram of algorithm computation from a $(l-1)$-level coloring κ to an "amended" l-level coloring $\hat{\kappa}$ (all at the same step t):

$$
\begin{array}{ll}
\mu_\kappa & \\
\quad \downarrow & \text{clustering} \\
\phi_\kappa & \\
\quad \downarrow & \text{extension: splitting in the proximity ball} \\
\varphi_\kappa & \\
\quad \downarrow & \text{minhash (under PNA)} \\
\mu_{\hat{\kappa}} &
\end{array}
$$

In each level $l \geq 1$, our goal is to color all intervals of density in a certain range $[n^{-\epsilon}, 1] \cdot d$ (whp), where $d \approx \frac{n/\lambda^t}{\beta^l}$, where $\beta = n^\epsilon$, as long as there are sufficiently many intervals of that density range overall[4]. At level $l = 0$, corresponding to the highest density for step t, we add an edge in G between the anchor A and each corresponding clustered interval I, and mark (fraction of) I as "already-matched" with the color $\perp$. In other levels, we color intervals found through extensions of clusters using φ.

The remaining case — "sparse" intervals we did not color via an anchor cluster extension as above — will be addressed by the careful use of $\mathbf{u}$-color, described next. We remark that, at the end of step t, there may be left some pairs of small densities which we still could not color, and are left $\mathbf{u}$-colored, and we will show a bound on those as well.

Controlling sparse sections: the u-color. In order to carry out the level-by-level coloring, we use the special color $\mathbf{u}$ (for un-colored). This color should be thought of as a "part" in the partition as well. At the beginning of a step, at level $l = 0$, all (fractions of) intervals which are not "already-matched" are assigned the $\mathbf{u}$ color, and (fractions of) intervals are moved from $\mathbf{u}$-color to "standard" colors $\in [\lambda^t]$ as levels progress. In particular, the $\mathbf{u}$-color helps with three aspects. First, it provides a way to track *sparse intervals* which cannot yet be colored and hence left pending for future levels. Second, if some sparse sections of intervals are never colored in the current step, then these intervals will remain $\mathbf{u}$-colored (and hence, at the end of the current step, form a part that is also bounded in size). Third, and more importantly, it allows us to "group together" sparse sections of intervals that are far apart (in their starting index).

In particular, such grouping of far intervals is done using the aforementioned "proximity balls", formally defined via Λ-*balls*: $\Lambda^\zeta_\kappa(I)$ is the smallest interval ball around I containing $\zeta \approx \beta^l$ $\mathbf{u}$-colored ℓ_1-mass on both left and right of I. Note that Λ-balls can contain a significantly larger set of fractions of intervals than ζ (if the in-between intervals are mostly colored $\neq \mathbf{u}$). At the same time, the ball contains at most 2ζ mass of $\mathbf{u}$-colored intervals, meaning that the potential $\phi(I)$ is distributed to a mass $\mu \leq 2\zeta$ of intervals,

[2]We'll often just use the verb "color" to describe that process.

[3]More precisely, to the fraction of I that shares a specific sampled color χ with A.

[4]Notice there can be multiple matches in a ball, hence the quantity we care about (and bound) is the *relative density* which is the ratio between "global" and "local" densities.

ensuring that, were I to be "corrupted" (e.g., $\pi[I] = \perp$, or $I, \pi[I]$ happen to be already separated), we will distribute $O(1)$ total *corrupted potential* to φ's of intervals in the Λ-ball of I.

To showcase the use of $\mathbf{u}$-color, consider again the running example introduced in Section III-A. In the early steps t, our algorithm will first color the dense intervals (i.e., via clustering/proximity balls, at lower levels l), leaving the sparse sections mostly unaffected, all colored in $\mathbf{u}$ (during such early steps, the dense intervals are partitioned into progressively smaller parts while most of the sparse ones remain $\mathbf{u}$-colored). Now consider a step t where the part sizes so far are $\lesssim n/\lambda^{t-1} \approx \Delta n$ and we sample $\gtrsim 1/\Delta$ anchors. At the lower levels l, the dense intervals will be partitioned further (continuing the process from the previous steps) and assigned a color $\chi \in [\lambda^t]$. However, when we reach the high levels l, and $\zeta \approx \beta^l = n^{\epsilon l}$ is close to Δn, some fraction of the sparse intervals will be clustered. Furthermore, since at that point the dense intervals are already colored (and have little $\mathbf{u}$-color), the Λ-balls around the clustered sparse intervals will be wide and cover most of the $\mathbf{u}$-colored sparse intervals. That allows us to finally partition the sparse intervals into smaller parts as well.

Keeping track of errors: Corruption. To measure and bound errors, in particular, intervals that do not match successfully, we use the notion of "corruption". First we define what it means for a (interval, color) pair to be corrupted. In the below, we use the distance function $\mathtt{dd}_F(a,b) = a \cdot \mathbb{1}[a > Fb]$, which can be thought of as a "robust" version of ℓ_1, which tolerates a "distortion" $F \gg 1$ (used for the non-PNA case). For now ignore parameter F, which will be clarified later.

Definition III.1 (Corrupted pairs)**.** *Fix alignment $\pi \in \Pi$, interval $I \in \mathcal{I}$, distortion $F \geq 1$, and graph G on $\mathcal{I}$. For color $\chi \in \nu$ in coloring κ, we say (I, χ) is a* (F, π, G)-corrupted pair *if any of the following holds:*

1) $\pi[I] = \perp$*;*
2) $\mathtt{ad}(I, \pi[I]) > c$*;*
3) $\chi \neq \perp$ *and* $\mathtt{dd}_F(\mu_\kappa(I,\chi), \mu_\kappa(\pi[I],\chi)) > 0$*; or*
4) $\chi = \perp$ *and* $(I, \pi[I])$ *are at hop-distance* > 2 *in* G*.*

For each interval $I \in \mathcal{I}$, we also define *corruption parameter* $\xi_F^{\kappa,\pi,G}(I) \in [0,1]$ as follows:

$$\xi_F^{\kappa,\pi,G}(I) = \sum_{\chi:(I,\chi) \text{ is } (F,\pi,G)\text{-corrupted pair}} \mu_\kappa(I,\chi). \quad (1)$$

In particular, an interval I is fully corrupted (in a coloring κ) if it does not have its π-matchable counterpart; and otherwise I is corrupted by the total ℓ_1 color-mass of $\mu_\kappa(I,*)$ where there is insufficient corresponding mass in $\mu_\kappa(\pi[I],*)$ (intuitively, the distribution of colors is too different). While our statements hold for any alignment in Π, we only care about a fixed optimal alignment π, and a single graph $G = G_{w,c}$. Hence, for ease of exposition, we say I (and $\pi[I]$) are *F-corrupted pair* and the corruption is $\xi_F^\kappa(I) \triangleq \xi_F^{\kappa,\pi,G}(I)$. Our main goal is to bound the *growth* of the total corruption $\xi_F^\kappa \triangleq \xi_F^\kappa(\mathcal{I})$ for each level/step by a constant factor. Our algorithm runs for a constant number of levels/steps, and hence finishes with ξ_F^κ which is proportional to the number of intervals without a π-matchable counterpart (starting corruption), upper-bounded by $O(\frac{w}{c} \cdot \mathtt{ed}(x,y))$. Also, at the end of the interval matching algorithm, all intervals are "already matched", i.e., all mass is on $\mu_\kappa(\mathcal{I}, \perp)$.

To bound the corruption growth, we also introduce the parameter $\rho(I)$, which measures the "local amount of corruption" of an $\mathbf{u}$-colored interval, based on the nearby corrupted intervals. In particular, $\rho(I)$ is defined for a Λ-ball around I as the ratio of the corruption to the $\mathbf{u}$-mass inside the Λ-ball. One can observe that for any fixed ζ radius of Λ, the sum of ρ over $\mathbf{u}$-colored intervals is proportional to the total sum of corruption.

Completing the algorithm under the perfect neighborhoods assumption (PNA). Once we compute the palettes φ of all intervals (as a function of the sampled anchors), we then use them to update μ for the next level. For illustrative purposes, we now complete the algorithm under the PNA, although our general algorithm will differ significantly from the PNA one. Recall that under PNA, all intervals are either at $\mathtt{ad}$-distance $\leq c$ or $\gg c$, and hence the intervals form equivalence classes according to their c-neighborhood.

Under PNA, we are guaranteed that the uncorrupted π-matchable pairs will get similar potentials —in fact, $\phi(I)$ and $\phi(\pi[I])$ are precisely equal (and non-zero whenever they are clustered by an anchor). More importantly, if we consider the φ palettes of $I, \pi[I]$, which gather the contributions from clusters containing $I, \pi[I]$ in their proximity ball, then one can prove that (the average) ℓ_1 distance between the two φ palettes is bounded as a function of the "local corruption", namely $\rho(I)$ and $\rho(\pi[I])$.

Using the ℓ_1-distance property of φ, we can assign a single color $\chi \in \nu$ to each interval (i.e., $\mu_\kappa(I,*)$ has support one), obtaining disjoint partitions. To generate such a color (for each interval) we can use a random *weighted min-wise hash function* $h \sim \mathcal{H}$ for $\mathbb{R}^\nu$ (say, using [32]) and use it to partition the vectors $\varphi(I,*)$ of all $\mathbf{u}$-colored intervals I. Specifically, sample a minhash $h : \mathbb{R}^\nu \to \nu$ and set the updated coloring to be $\mu_{\widehat{\kappa}}(I, h(\varphi(I))) \leftarrow 1$ (the rest are 0) for all I for which $\mu_\kappa(I, \mathbf{u}) = 1$ at the end of the previous level.

For a glimpse of the analysis, recall that our overall goal is to make each part in partition smaller (for complexity) with only a constant-factor corruption growth (for correctness); also, we care only to partition areas with large mass of intervals with density in some range $[n^{-\epsilon}, 1] \cdot d$ (per level). To control the size of parts, we cannot afford to assign a single

color to too many intervals; hence we drop from φ all colors of potential below some fixed threshold $o(n^{-\epsilon})$, ensuring we keep each color in the palettes φ of at most $\frac{n^{1+O(\epsilon)}}{\lambda^{t-1}}$ intervals. For bounded corruption, we note that minhash gives us a bound proportional to the Jaccard distance between $\varphi(I)$ and $\varphi(\pi[I])$, while the bound we have is over ℓ_1. This is where the $\mathbf{u}$-color will eventually help. First, consider an interval I such that $\Omega(\epsilon)$ portion of its proximity ball Λ is composed of intervals of density $\in [n^{-\epsilon}, 1] \cdot d$, where Λ is of radius ζ. Then the palette $\varphi(I)$ has ℓ_1 mass $\Omega(\epsilon)$ whp after thresholding since: 1) some intervals in Λ will be clustered whp (they are dense enough), and 2) once clustered, the generated potential is large enough to pass the threshold (since they are not too dense). In this case, we are done (without using the color $\mathbf{u}$): the Jaccard distance is proportional to the ℓ_1 distance between $\varphi(I)$ and $\varphi(\pi[I])$, and hence the probability of separating I from $\pi[I]$ is bounded by the "local corruption" (which overall is bounded by the total corruption). Second, consider the case when $1 - o(\epsilon)$ portion of intervals in the Λ ball are outside the aforementioned density range. Then we add $\mathbf{u}$ to φ of intervals in the ball Λ, filling it up to reach $\|\varphi(I)\|_1 = \Omega(\epsilon)$ — this will mean that such intervals are likely to mostly map to $\mathbf{u}$ (this increases corruption by a factor ≤ 2). This process also guarantees Λ balls at the next level have $(1 - o(\epsilon l)) \cdot \zeta$ mass of *sparse intervals*, of density $\lesssim \frac{n/\lambda^t}{\beta^l}$.. One can then prove that, after running this process for $\approx 1/\epsilon$ levels, the set of intervals corresponding to each color, including $\mathbf{u}$, is of size $\lesssim n/\lambda^t$ only.

Since we could not directly extend the minhash construction to the general non-PNA case, we do not present this construction in the paper, but rather use it as an intuition for the more "robust" version as we describe next.

D. Imperfect neighborhoods

To eliminate the perfect neighborhood assumption (PNA), we must rely on the weaker form of transitivity instead, from the triangle inequality: $\mathcal{N}_c(I) \subseteq \cup_{J \in \mathcal{N}_c(I)} \mathcal{N}_{2c}(J) \subseteq \mathcal{N}_{3c}(I)$. Note that the usual ideas to deal with such "weaker transitivity" do not seem applicable here. E.g., if we pick the threshold of "close" to be random $\in [c, O(c)]$, there's still a constant probability of separating $I, \pi[I]$. One could instead apply the more nuanced metric random partitions, such as from [33], which would partition the metric $(\mathcal{I}_w, \mathtt{ad}_w)$ (thus putting us back into the perfect neighborhood assumption), with the probability of $I, \pi[I]$ ending up in the same part being $\geq n^{-\epsilon}$ — which has been useful in other contexts by repeating such partition $\approx n^{\epsilon}$ times. However, such a process results in a random partition retaining only $n^{1-O(\epsilon)}$ π-matched pairs, which is not enough to reconstruct even those matched pairs (intuitively, the strings are "too corrupted", as if the edit distance is $(1 - o(1)) \cdot n$), making it inapplicable for our subtle application (here again, this challenge would not be a big issue if additive approximation were allowed). Overall, dealing with imperfect neighborhoods proved to be a substantial challenge for us, and we develop several first-of-a-kind tools specifically to deal with it.

Eventually, we still sample a cost c_i from an ordered set $E_c = \{c_1, c_2, \ldots\} \subset [c, O(c)]$. We will want that the sampled cost satisfies that $\mathcal{N}_{c_i}(\mathcal{N}_{c_i}(I)) \subseteq \mathcal{N}_{c_{i+1}}(I)$. To ensure the latter, the set E_c of costs is exponentially-growing, i.e., $c_i = O(c) \cdot 3^i$.

Distortion Resilient Distance. Note that we may assign somewhat different ϕ potential to I and $\pi[I]$ from the sampling procedure above. The *distortion* (ie, the multiplicative difference) between $\phi(I)$ and $\phi(\pi[I])$ for π-matchable pairs can be as high as n^{α} for some constant α. Such distortion makes it impossible to obtain an ℓ_1 bound on φ which is proportional to ρ. Instead, we deal with such distortion by employing a *distortion resilient* version of ℓ_1.

Definition III.2. *Fix $p, q \in \mathbb{R}^n_+$. We define the F-*distortion resilient distance*,*

$$\mathtt{dd}_F(p, q) = \sum_{i : p_i > F \cdot q_i} |p_i|$$

This function allows us to define and control *corruption* of (interval, color) pairs, by differentiating distortion (which captures multiplicative errors) from corruption (which captures additive ones). As part of our analysis, we will show several basic properties of dd distance and develop dd-*preserving soft-transformations*, that will replace the hard thresholds of the minhash construction. Intuitively, dd replaces the use of the ℓ_1/Jaccard metric on the vectors ϕ, which is a key enabler for using minhash. However, dd is not a metric in any reasonable sense, rendering the minhash construction obsolete (e.g., it seems unreasonable to expect any kind of LSH under dd).

Assigning potential to (interval, color) pairs. Since maintaining equivalence classes is essential for our construction, we analyze pairs of interval and colors in $\mathcal{I} \times \nu$ (which, combinatorially, can be thought of as "fractions of intervals"). Thus when we add some new ϕ-potential to a clustered pair (I, χ'), we multiply such increment by it's μ mass, meaning we add potential $\approx \mu(I, \chi') \frac{n}{d \cdot \lambda^t}$. Similarly, splitting the ϕ potential to $\mathbf{u}$-colored-pairs in Λ balls (i.e., assigning φ) is done in a pro-rated fashion, weighted according to respective $\mu(\cdot, \mathbf{u})$ masses.

Assigning u potential: pivot sampling. While so far we discussed assigning non-$\mathbf{u}$ colors in φ of (fractions of) intervals in a level, we also need to discuss how to assign $\mathbf{u}$-color in φ_κ and, eventually, amended coloring $\mu_{\widehat{\kappa}}$. It may be tempting to merely subtract the assigned fraction of non-$\mathbf{u}$-color from the $\mathbf{u}$-color mass, but this would result in additive errors (for the color $\mathbf{u}$), while dd only allows for multiplicative distortion. Since we need $\mathbf{u}$ colors to agree up to a fixed distortion as well, we compute the new $\mathbf{u}$-color mass directly, via a different technique for explicitly

measuring *sparseness* (which is the central purpose of $\mathbf{u}$-color). To accomplish this measurement, we developed a procedure called *pivot sampling*, which somewhat resembles the way we assign potentials for the other colors. First, we downsample $\mathcal{I} \times \nu$ into a smaller set of *pivots* $\mathcal{V}$. Second, we *approximate the density* of each pivot in $\mathcal{V}$, for each possible cost E_c, thus generating θ-*potential* scores for each pivot. Third and last, such θ potential is split among the intervals in a Λ-ball in a similar fashion to how we split ϕ, generating $\varphi(\cdot, \mathbf{u})$ potentials to *intervals in sparse areas*. This rather involved process, specific to dealing with imperfect neighborhoods, requires much care as we need to control: (1) corruption of $\mathbf{u}$-colors; (2) balance of palettes φ (as we describe next); (3) sparsity guarantees for $\mathbf{u}$-color at the end of each step t; and (4) computational efficiency of such sampling mechanism.

Amending a coloring in a level, using φ. While dd is a convenient analytical tool for bounding corruption, it lacks the basic properties to allow coordinated sampling between π-matchable pairs (e.g., it is not even symmetric). Instead of sampling a color from $\varphi(I, *)$ (as was done under PNA), we add *all* colors in $\varphi(I, *)$ to the amended coloring $\mu_{\widehat{\kappa}}(I, *)$. To maintain a distribution of colors, we first combine φ with pre-existing non-$\mathbf{u}$-colors in μ, and then normalize to have ℓ_1-mass of $\mu_\kappa(I, \mathbf{u})$, i.e., what "remains to be colored" (i.e., ensuring that the overall amended $\mu_{\widehat{\kappa}}(I, *)$ is a distribution). As in the PNA case, we need to bound extra *corruption from normalization* by ensuring that the palettes φ have constant norms. While this analysis for the PNA solution is immediate (by construction), here, instead, we employ several combinatorial arguments that analyze mass of pairs with certain density over certain set of costs, eventually showing that at each level, we either add sufficient clustered-colors or $\mathbf{u}$-colors to all intervals while maintaining guarantees (1)–(4) above.

Controlling growing distortion. Our arguments require that throughout the matching phase, the dd distortion F is bounded by $n^{o(\epsilon)}$ (in particular, to maintain control over the aforementioned soft-transformations). Many of our algorithmic steps generate extra distortion. To control both distortion (multiplicative error) and corruption (additive error), we parametrize maximum distortion $F = F(t, l)$ for each step/level a priori, and bound corruption ξ_F^κ at each step/level using the pre-determined distortion parameter $F = F(t, l)$. The final approximation factor is a function of the maximum cost in $\frac{1}{c}E_c$ (which is further determined by the "base distortion" $F(1, 0) = n^\alpha$), together with the corruption factor we show in each step. At the end of the day, a distortion F bounded by $n^{o(\epsilon)}$ allows us to carry out the above arguments (i.e., some of the above arguments can only work under small distortion F).

E. The metrics $\mathtt{ad}_w$

We now briefly discuss the algorithm for computing the $\mathtt{ad}_w(I, J)$ distance, using oracle calls to $\mathfrak{D}_{w/\gamma}$ metric. This metric is used to compute distances when building the graph G_w, in the Interval Matching algorithm. Note that the latter makes $n^{1+O(\epsilon)}$ oracle calls to ad, and hence the algorithm has to run in time $n^{O(\epsilon)} = \mathrm{poly}(\gamma)$.

Intuitively, $\mathtt{ad}_w(I, J)$ is meant to capture the following distance, which should be thought of as an extension of the edit distance to alphabet with the metric $(\mathcal{I}_{w'}, \mathfrak{D}_{w'})$ where $w' = w/\gamma$:[5]

$$\min_\pi \sum_{i \in [w]} \tfrac{1}{w'} \mathfrak{D}_{w'}(I + i, J + \pi(i)),$$

where π ranges over all alignments of indexes of I to indexes of J. It's not hard to see that, if $\mathfrak{D}(I, J) = \mathtt{ed}(I, J)$, then, for $I = X_{i,w}, J = Y_{j,w}$, the above distance is between $\mathtt{ed}(X_{i,w}, Y_{j,w})$ and $\mathtt{ed}(X_{i,2w}, Y_{j,2w})$.

However, this distance function is hard to compute fast: not only it is as hard as computing edit distance on w-length strings, but even linear time (in $w \gg \mathrm{poly}(\gamma)$) is too much for us. In particular, it does not use the fact that $\mathfrak{D}_{w'}(I + i, J + \pi(i))$ captures the information of blocks of length w'. Hence, it is natural to approximate the above by considering a "rarefication" of the above sum as follows:

$$\mathtt{ad}_w(I, J) = \min_\pi \sum_{i \in [\gamma]} \tfrac{1}{w'} \mathfrak{D}_{w'}(I + iw', J + \pi(iw')). \quad (2)$$

However, the latter will not satisfy the triangle inequality — which is crucial in the Interval Matching Algorithm — and in fact is not even symmetric: e.g., if the optimal $\pi(i) = i+1$, the $\mathtt{ad}_w(J, I)$ would be using $\mathfrak{D}$ on completely different arguments. This is especially an issue since certain $\mathfrak{D}$ pairs may substantially over-estimate ed (and hence "shift by one" can change the distance a lot).

Indeed, ensuring triangle inequality is the main challenge for defining and computing $\mathtt{ad}_w$ here. We manage to define an appropriate distance ad, satisfying triangle inequality for "one scale only" metrics $\mathtt{ad}_{w,c}$, designed for distances in the range $\approx [c, \gamma c]$, which turns out to be enough for the Interval Matching algorithm.

First, we note that we have two different algorithms: for $c \le w/\gamma$, and $c > w/\gamma$. The reason there's a big difference between the two cases is that when $c > w/\gamma$, the alignment π may have a large displacement $|i - \pi(i)| \ge w/\gamma$, bigger than the length of "constituent" intervals for which we have the base metric $\mathfrak{D}_{w'}$. Hence, for the "large distance" regime, when $c \ge w/\gamma$, we uses a slightly different (and simpler) algorithm that runs in time $\approx \mathrm{poly}(w/c)$, and hence is only good when c is sufficiently large.

[5] $I + i$ means the interval starting i positions to the right of the start of I.

Finally, we sketch the harder, $\mathrm{poly}(\gamma)$-time algorithm, for not-large c. The idea is to allow alignment shifts in both intervals. More formally, let $\mathcal{A}$ be the set of functions $A = (A_x, A_y)$ where $A_x, A_y : [-\gamma, \gamma) \to [T]$ are non-decreasing functions with $A_x[-\gamma] = A_y[-\gamma] = 0$ and $A_x[\gamma - 1] = A_y[\gamma - 1]$, and where $T = \gamma^2$. We define the distance $\mathtt{ad}_{w,c}(I, J)$, to be (essentially), where $\theta = \Theta(c/w)$:

$$\min_{A \in \mathcal{A}} \sum_{i \in [-\gamma,\gamma)} \frac{1}{T} \sum_{\Delta \in [3T - \|A[i]\|_1]} \mathfrak{D}_{w'}(\quad I + w'(i + \theta(\Delta + A_x[i])), \\ J + w'(i + \theta(\Delta + A_y[i])) \\ + (A_x[\gamma] + A_y[\gamma])\theta w'.$$

Intuitively, ignoring Δ-sum (i.e., think $\Delta = 0$), we obtain an alignment of I to J where the starting positions (of w'-length intervals) are close to multiples of w' in *both* strings (as opposed to only one string, as in Eqn. (2)). While allowing such an alignment is enough for ensuring symmetry, it is still not enough to ensure triangle inequality: think of the case when $\mathtt{ad}(I, J)$ and $\mathtt{ad}(J, K)$ use the maximally-allowed values of the alignment (namely γ^2), in which case $\mathtt{ad}(I, K)$ cannot use the natural composition of the two alignments (since it's out of bounds). This is where the summation over Δ saves the day (and is another instance of "averaging it out"). The last definition can also be computed in $\mathrm{poly}(\gamma)$ time by a standard dynamic programming.

Finally, we remark that, at the end of the day, we cannot guarantee a per-pair upper bound on $\mathtt{ad}(I, J)$, but only on average, and only when comparing $X_{i,w}$ against $Y_{\pi(i),w}$ (although the triangle inequality is true everywhere). This is, nonetheless, just enough for computing $\mathtt{ed}(x, y)$ in the end.

IV. Top-Level Algorithm

We now describe our "top-level" algorithm. We assume here that the first $(1-o(1))n$ positions of x and y are equal; we can remove this assumption by padding x, y with some fixed unique character \$, increasing the size of x, y by a factor of, say, $O(\log n)$.

Our algorithm consists of $\log_\gamma n$ iterations, where $\gamma = n^\epsilon$. For each $w = \gamma^i$, $i \in [\log_\gamma n]$, we construct the metric $(\mathcal{I}_w, \mathfrak{D}_w)$, which approximates the metric $(\mathcal{I}_w, \mathtt{ed}_w)$ in a certain average sense, for most of the "relevant" pairs $I, J \in \mathcal{I}_w$. Each iteration consists of two components: the alignment distance algorithm, and the interval matching algorithm, described in later sections.

Alignment Distance algorithm. Assuming fast access to $(\mathcal{I}_{w/\gamma}, \mathfrak{D}_{w/\gamma})$, the metric constructed at the previous iteration, our alignment algorithm is an oracle for computing the distance $\mathtt{ad}_w(\cdot, \cdot)$ on w-length intervals. More specifically, we have $O(\log n)$ such distance measures, $\mathtt{ad}_{w,c}$, each for a target cost scale $c \in S_w$. Each such function $\mathtt{ad}_{w,c}(\cdot, \cdot)$ evaluation is an edit-distance-like dynamic programming of size $\mathrm{poly}(\gamma)$, and overall can be computed using $\mathrm{poly}(\gamma) = n^{O(\epsilon)}$ time and number of oracle calls to $\mathfrak{D}_{w/\gamma}$.

Note that this algorithm is not run directly, but instead is used as an oracle inside the matching algorithm described next.

Interval Matching algorithm. We construct a weighted graph G_w on $\mathcal{I}_w$, such that the shortest distance approximates the $\mathtt{ad}_w$ distance on intervals. Again, this won't be achieved for all pairs of intervals, but only for interval pairs that "matter", i.e., that are in an optimal alignment for $\mathtt{ed}(x, y)$. The graph G_w is union of graphs $G_{w,c}$, for $c \in S_w = \{0, 1, 2, 4, \ldots w\}$, each of them approximating $\mathtt{ad}_{w,c}$ at "scale c". Constructing the graphs $G_{w,c}$ is the heart of the matching algorithm.

Once we have the graph G_w, we build a fast distance oracle data structure on it, using Theorem IV.1 below, and whose output is the desired metric $\mathfrak{D}_w$. Overloading the notation, we call $\mathfrak{D}_w$ both the distance oracle data structure as well as the metric it produces.

Theorem IV.1. *[29] For any constant integer $\alpha \geq 3$, and given any weighted graph G on n nodes and m edges, we can build a distance oracle data structure with the following properties:*

- *supports distance queries: given u, v outputs $\mathfrak{D}_G(u, v)$ which is a α-factor approximation to the shortest path distance between u, v in the graph;*
- *$\mathfrak{D}_G(u, v)$ is a (n-point) metric;*
- *runtime per query is $\tilde{O}(n^{2/\alpha})$;*
- *data structure uses $\tilde{O}(n^{1+2/\alpha})$ space, and pre-processing time is $\tilde{O}(mn^{2/\alpha})$.*

Top-level algorithm is described in Algorithm 1. At the beginning, when $w = \gamma$, we use the metric $(\mathcal{I}_1, \mathfrak{D}_1)$, which is just the metric on all positions in x and y, where two positions are at distance 0 iff the positions contain the same character, and 1 otherwise. At the end, when $w = n$, we can extract the distance between x and y, which is our final approximation.

Algorithm 1 EstimateEditDistance(x, y, ϵ, n)

function ESTIMATEEDITDISTANCE(x, y, ϵ, n)

Fix C_m to be the constant from Theorem IV.2.

$\gamma \leftarrow n^{\zeta\epsilon}$, for a small absolute constant ζ.

$\mathfrak{D}_1$ is a data structure that, given two positions into x and/or y, outputs 0 iff the characters in those positions are equal and 1 otherwise.

for $w \in \{\gamma, \gamma^2, \dots, n\}$ **do**

Let $\mathcal{X}_w, \mathcal{Y}_w$ be sets of all w-length intervals on x-axis and y-axis respectively, with $\mathcal{I}_w = \mathcal{X}_w \cup \mathcal{Y}_w$.

for $c \in S_w$ **do**

Initialize new coloring κ of a single color assigned with mass 1 to all $\mathcal{I}_w$.

$G_{w,c} \leftarrow$ MATCHINTERVALS$(\mathfrak{D}_{w/\gamma}, c, \kappa, 1)$.

end for

$G_w = \cup_{c \in S_w} C_m \cdot c \cdot G_{w,c}$ and add edges $(X_{i,w}, X_{i+1,w}), (Y_{i,w}, Y_{i+1,w})$ with unit cost for all i.

$\mathfrak{D}_w \leftarrow$ data structure from Theorem IV.1 on graph G_w for approximation $10/\epsilon$.

end for

return $\mathfrak{D}_n(X_{i,n}, Y_{i,n})$ for a randomly chosen $i \in [n]$.

end function

A. Main guarantees

The guarantees of the algorithm will follow from the following two main theorems.

Theorem IV.2 (MATCHINTERVALS)**.** *Fix $\epsilon > 0$, $w, n \in \mathbf{N}$, and an alignment $\pi \in \Pi$, as well as cost $c \in S_w$. Suppose the distance $\mathtt{ad}_{w,c}$ is a metric, for which we have query access running in time $T_{\mathtt{ad}}$. Then, the algorithm* MATCHINTERVALS *builds an undirected graph $G_{w,c}$, over intervals $\mathcal{I}_w = \mathcal{X}_w \cup \mathcal{Y}_w$, such that:*

1) *For all edges $(I, J) \in G_{w,c}$, we have $\mathtt{ad}_{w,c}(I, J) \le C_m \cdot c$, where $C_m = C_m(\epsilon)$ is a constant.*
2) *There exists a set of at most $O(k_c)$ indices $i \in [n]$, such that*[6] $\mathtt{ad}_{w,c}(X_{i,w}, Y_{\pi(i),w}) \le c$ *and also* $\mathrm{dist}_{G_{w,c}}(X_{i,w}, Y_{\pi(i),w}) > 2$*, where $k_c = |\{i \mid \mathtt{ad}_{w,c}(X_{i,w}, Y_{\pi(i),w}) > c\}|$, and $\mathrm{dist}_{G_{w,c}}$ is the hop-distance in $G_{w,c}$.*
3) *The runtime of the algorithm is $O(T_{\mathtt{ad}} \cdot n^{1+O(\epsilon)})$.*

As described above, using the algorithm from the above theorem, we build a graph G_w, which is the union of scale graphs $G_{w,c}$. Then we take $\mathfrak{D}_w$ to be the fast distance oracle of the shortest path on the graph G_w, using Theorem IV.3.

Next theorem says that, given access to $\mathfrak{D}_{w/\gamma}$, we can compute $\mathtt{ad}_w(I, J)$ for any two intervals $I, J \in \mathcal{I}$, which corresponds to the "natural extension" of $\mathfrak{D}$ from length-w/γ to length-w substrings.

Theorem IV.3 (alignment distance $\mathtt{ad}$)**.** *Fix w and $w' = w/\gamma$, and suppose we have a data structure for a metric $\mathfrak{D}_{w'}$ that satisfies the following for some constant $C \ge 1$, and any $i, j \in [n]$: 1) $\mathfrak{D}_{w'}(X_{i,w'}, Y_{j,w'}) \ge \mathtt{ed}(X_{i,w'}, Y_{j,w'})$, 2) $\mathfrak{D}_{w'}(X_{i,w'}, X_{i+1,w'}) \le C$ (and same for Y intervals), and 3)*

$$\min_{\pi \in \Pi} \sum_{i \in [n]} \tfrac{1}{w'} \mathfrak{D}_{w'}(X_{i,w'}, Y_{\pi(i),w'}) \le C \cdot \mathtt{ed}(x, y).$$

Then, for any $c \in S_w$, there is an algorithm defining $\mathtt{ad}_{w,c}(\cdot,\cdot)$ with the following properties:

- *$\mathtt{ad}_{w,c}$ is a metric;*
- *$\mathtt{ad}_{w,c}(X_i, Y_j) \ge \min\{\mathtt{ed}(X_i, Y_j), c\sqrt{\gamma}\}$;*
- *if we define $\mathtt{ad}_w(X_i, Y_j) \triangleq \sum_{c \in S_w} c \cdot \mathbb{1}[\mathtt{ad}_{w,c}(X_i, Y_j) > c]$, we have:*

$$\min_{\pi \in \Pi} \sum_{i \in [n]} \tfrac{1}{w} \mathtt{ad}_w(X_{i,w}, Y_{\pi(i),w}) \le O(C) \cdot \mathtt{ed}(x, y). \quad (3)$$

- *for any two intervals $I, J \in \mathcal{I}_w$, $\mathtt{ad}_{w,c}(I, J)$ can be computed using $\tilde{O}(\gamma^{O(1)})$ time and $\mathfrak{D}_{w'}$ queries.*

We remark that $\mathtt{ad}_w$ is not guaranteed to be a metric, which is the reason why we use $\mathtt{ad}_{w,c}$ in the theorem statement.

B. Proof of Theorem I.1

Proof of Theorem I.1 follows from the above two theorems, IV.2 and IV.3. In particular, the inductive hypothesis is that, for $w = \gamma^i$, where $i \in [\log_\gamma n]$, we have that the distance oracle data structure $\mathfrak{D}_w$ outputs a metric $\mathfrak{D}_w$ with the following properties, for some constant $C_w = C(\epsilon, \log_\gamma w)$, whp:

1) $\mathfrak{D}_w(X_{i,w}, Y_{j,w}) \ge \mathtt{ed}(X_{i,w}, Y_{j,w})$ for any $i, j \in [n]$;
2) $\mathfrak{D}_w(X_{i,w}, X_{i+1,w}) \le 10/\epsilon$ (and same for Y intervals);
3) $\min_{\pi \in \Pi} \sum_{i \in [n]} \frac{1}{w} \mathfrak{D}_w(X_{i,w}, Y_{\pi(i),w}) \le C_w \cdot \mathtt{ed}(x, y)$.

Base case: for $w = 1$, this is immediate by construction of $\mathfrak{D}_1$.

Now assume the inductive hypothesis for $w' = w/\gamma$ and we need to prove it for w. By inductive hypothesis, $\mathfrak{D}_{w'}$ satisfies hypothesis of Theorem IV.3, and hence we can apply it to obtain an oracle query to metrics $\mathtt{ad}_{w,c}$; each oracle query takes $O(\gamma^{O(1)})$ time. Let π be the optimal alignment obtained from Theorem IV.3.

Define $\tau(\cdot,\cdot)$ to be the distance in the graph G_w constructed in the algorithm. We will prove below that τ is a metric satisfying the above properties. Hence, once we build a fast distance oracle $\mathfrak{D}_w$ on the graph G_w (using Theorem IV.1 with $\alpha = \Theta(1/\epsilon)$), its output metric $\mathfrak{D}_w$ satisfies $\tau \le \mathfrak{D}_w \le O(\tau)$, and hence the inductive hypothesis.

To prove the first property of the inductive hypothesis, consider any two intervals I, J, and the shortest path $v_1, \dots v_k$ between them, where $v_1 = I$ and $v_k = J$. We have that (v_i, v_{i+1}) is an edge in some graph G_{w,c_i}, or is an extra edge of cost 1; call E the set of the latter i's. Hence the cost $\tau(I, J) = \sum_{i \notin E} C_m c_i + |E|$. For $i \notin E$, by Theorem IV.2 and Theorem IV.3, we have that

[6] Recall from the preliminaries that $\mathtt{ad}_w(X_i, Y_\perp) = w$.

$C_m c_i \geq \mathtt{ad}_{w,c_i}(v_i, v_{i+1}) \geq \mathtt{ed}(v_i, v_{i+1})$ (note that the other part of the min cannot happen). Also, for $i \in E$, we have that $1 = \tau(v_i, v_{i+1}) \geq \mathtt{ed}(v_i, v_{i+1})$ as $\mathtt{ed}(X_i, X_{i+1}) \leq 1$ (and same for Y's). Hence $\mathfrak{D}(I, J) \geq \tau(I, J) = C_m \sum_{i \notin E} c_i + |E| \geq \mathtt{ed}(v_1, v_2) + \ldots + \mathtt{ed}(v_{k-1}, v_k) \geq \mathtt{ed}(I, J)$.

The second property is immediate by construction of the graph G_w and the fact that approximation of the distance oracle $\mathfrak{D}_w$ is taken to be $10/\epsilon$.

For the third property, we note that $\tau(I, J)$ is upper bounded by $\sum_{c \in S_w} 4 \cdot C_m c \cdot \mathbb{1}[\mathrm{dist}_{G_{w,c}}(I, J) > 2]$. Hence:

$$\sum_{i \in [n]} \tau(X_{i,w}, Y_{\pi(i),w}) \leq \sum_{i \in [n]} \sum_{c \in S_w} 4C_m c \cdot \mathbb{1}[\mathrm{dist}_{G_{w,c}}(X_{i,w}, Y_{\pi(i),w}) > 2]$$

$$= 4C_m \sum_{c \in S_w} \sum_{i \in [n]} c \cdot \mathbb{1}[\mathrm{dist}_{G_{w,c}}(X_{i,w}, Y_{\pi(i),w}) > 2].$$

For fixed c, we have that, by Theorem IV.2:

$$\sum_{i \in [n]} \mathbb{1}[\mathrm{dist}_{G_{w,c}}(X_{i,w}, Y_{\pi(i),w}) > 2] \leq O(k_c)$$

$$= O\left(\sum_i \mathbb{1}[\mathtt{ad}_{w,c}(X_{i,w}, Y_{\pi(i),w}) > c] \right).$$

Therefore,

$$\sum_{i \in [n]} \tau(X_{i,w}, Y_{\pi(i),w}) \leq O\left(\sum_{c \in S_w} \sum_{i \in [n]} c \cdot \mathbb{1}[\mathtt{ad}_{w,c}(X_{i,w}, Y_{\pi(i),w}) > c] \right)$$

$$\leq O\left(\sum_{i \in [n]} \mathtt{ad}_w(X_{i,w}, Y_{\pi(i),w}) \right).$$

Altogether, we obtain, using Theorem IV.3 again together with the inductive hypothesis for $w' = w/\gamma$:

$$\sum_{i \in [n]} \mathfrak{D}_w(X_{i,w}, Y_{\pi(i),w}) \leq O(1) \cdot \sum_{i \in [n]} \tau(X_{i,w}, Y_{\pi(i),w})$$

$$\leq O(1) \cdot \sum_{i \in [n]} \mathtt{ad}_w(X_{i,w}, Y_{\pi(i),w}) \leq O(w \cdot \mathtt{ed}(x, y)),$$

completing the proof of the inductive hypothesis.

Now we argue that the final output produced by the top-level algorithm is a constant factor approximation. Consider the $\mathfrak{D}_w$ guarantees for $w = n$, and fix the minimizing π, and constant $C_n = C_w$. For a random index $i \in [n]$, with probability at least 0.9, we have that: 1) $i \in [1, n - o(n)]$, 2) $\pi(i) \neq \perp$, and $\mathfrak{D}(X_{i,n}, Y_{\pi(i),n}) \leq O(C_n) \cdot \mathtt{ed}(x, y)$. Furthermore note that $|i - \pi(i)| \leq C_n \cdot \mathtt{ed}(x, y)$, and hence, $\mathfrak{D}(X_{i,n}, Y_{i,n}) \leq \mathfrak{D}(X_{i,n}, Y_{\pi(i),n}) + \frac{10}{\epsilon} \cdot |i - \pi(i)| \leq O(C_n/\epsilon) \cdot \mathtt{ed}(x, y)$. Also, since $i \leq n - o(n)$, we have that $\mathfrak{D}(X_{i,n}, Y_{i,n}) \geq \mathtt{ed}(X_{i,n}, Y_{i,n}) = \mathtt{ed}(x, y)$.

Concluding, the algorithm produces a $O(C_n/\epsilon)$ approximation to $\mathtt{ed}(x, y)$, with probability ≥ 0.9. Note that C_n is a constant, depending on ϵ, as we have only a constant number of iterations, each incurring a constant factor approximation.

The runtime guarantee follows trivially from time guarantees of Theorems IV.2 and IV.3.

REFERENCES

[1] D. Chakraborty, D. Das, E. Goldenberg, M. Koucky, and M. Saks, "Approximating edit distance within constant factor in truly sub-quadratic time," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018, pp. 979–990.

[2] M. Kouckỳ and M. E. Saks, "Constant factor approximations to edit distance on far input pairs in nearly linear time," in *Proceedings of the Symposium on Theory of Computing (STOC)*, 2020, arXiv preprint arXiv:1904.05459.

[3] J. Brakensiek and A. Rubinstein, "Constant-factor approximation of near-linear edit distance in near-linear time," in *Proceedings of the Symposium on Theory of Computing (STOC)*, 2020, arXiv preprint arXiv:1904.05390.

[4] W. J. Masek and M. Paterson, "A faster algorithm computing string edit distances," *J. Comput. Syst. Sci.*, vol. 20, no. 1, pp. 18–31, 1980.

[5] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[6] S. C. Sahinalp, "Edit distance under block operations," in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Springer, 2008.

[7] A. Backurs and P. 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.

[8] A. Abboud, T. D. Hansen, V. V. Williams, and R. 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*. ACM, 2016, pp. 375–388.

[9] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.

[10] E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[11] G. M. Landau, E. W. Myers, and J. P. Schmidt, "Incremental string comparison," *SIAM J. Comput.*, vol. 27, no. 2, pp. 557–582, 1998.

[12] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar, "Approximating edit distance efficiently," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2004, pp. 550–559.

[13] T. Batu, F. Ergün, and C. Sahinalp, "Oblivious string embeddings and edit distance approximations," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2006, pp. 792–801.

[14] A. Andoni and K. Onak, "Approximating edit distance in near-linear time," *SIAM J. Comput. (SICOMP)*, vol. 41, no. 6, pp. 1635–1648, 2012, previously in STOC'09.

[15] R. Ostrovsky and Y. Rabani, "Low distortion embedding for edit distance," *J. ACM*, vol. 54, no. 5, 2007, preliminary version appeared in STOC'05.

[16] A. Andoni, R. Krauthgamer, and K. 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.

[17] T. Batu, F. Ergün, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld, and R. Sami, "A sublinear algorithm for weakly approximating edit distance," in *Proceedings of the Symposium on Theory of Computing (STOC)*, 2003, pp. 316–324.

[18] M. Boroujeni, S. Ehsani, M. Ghodsi, M. HajiAghayi, and S. Seddighin, "Approximating edit distance in truly subquadratic time: Quantum and mapreduce," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2018, pp. 1170–1189.

[19] E. Goldenberg, A. Rubinstein, and B. Saha, "Does preprocessing help in fast sequence comparisons?" in *Proceedings of the Symposium on Theory of Computing (STOC)*, 2020.

[20] A. Andoni, "Simpler constant-factor approximation to edit distance problems," 2018, manuscript, available at http://www.cs.columbia.edu/~andoni/papers/edit/.

[21] D. Chakraborty, D. Das, and M. Koucky, "Approximate online pattern matching in sub-linear time," in *FSTTCS*, 2019.

[22] E. Goldenberg, R. Krauthgamer, and B. Saha, "Sublinear algorithms for gap edit distance," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2019, pp. 1101–1120.

[23] A. Andoni and R. Krauthgamer, "The smoothed complexity of edit distance," *ACM Transactions on Algorithms*, vol. 8, no. 4, p. 44, 2012, previously in ICALP'08. [Online]. Available: http://doi.acm.org/10.1145/2344422.2344434

[24] W. Kuszmaul, "Efficiently approximating edit distance between pseudorandom strings," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2019, pp. 1165–1180.

[25] M. Hajiaghayi, S. Seddighin, and X. Sun, "Massively parallel approximation algorithms for edit distance and longest common subsequence," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2019, pp. 1654–1672.

[26] M. Hajiaghayi, M. Seddighin, S. Seddighin, and X. Sun, "Approximating lcs in linear time: Beating the barrier," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2019, pp. 1181–1200.

[27] A. Rubinstein, S. Seddighin, Z. Song, and X. Sun, "Approximation algorithms for lcs and lis with truly improved running times," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2019, pp. 1121–1145.

[28] A. Rubinstein and Z. Song, "Reducing approximate longest common subsequence to approximate edit distance," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2020, pp. 1591–1600.

[29] J. Matoušek, "On the distortion required for embedding finite metric spaces into normed spaces," *Israel Journal of Mathematics*, vol. 93, no. 1, pp. 333–344, Dec 1996. [Online]. Available: https://doi.org/10.1007/BF02761110

[30] M. Thorup and U. Zwick, "Approximate distance oracles," *J. ACM*, vol. 52, no. 1, pp. 1–24, Jan. 2005. [Online]. Available: http://doi.acm.org/10.1145/1044731.1044732

[31] S. Chechik, "Approximate distance oracles with constant query time," in *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '14. New York, NY, USA: ACM, 2014, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/2591796.2591801

[32] M. Charikar, "Similarity estimation techniques from rounding," in *Proceedings of the Symposium on Theory of Computing (STOC)*, 2002, pp. 380–388.

[33] M. Mendel and A. Naor, "Ramsey partitions and proximity data structures," *Journal of the European Mathematical Society*, vol. 9, no. 2, pp. 253–275, 2007, extended abstract appeared in FOCS 2006.