LOCALITY SENSITIVE HASHING, GAP EDIT DISTANCE, AND SIMILARITY FOR SEQUENCE DATA APPLICATIONS

BILL SUN (BYS2107@COLUMBIA.EDU) , NICHOLAS DEAS (NID2107@COLUMBIA.EDU) , AND SEI CHANG (SHC2170@COLUMBIA.EDU)

May 15, 2023

1. Introduction. Techniques such as Locality Sensitive Hashing (LSH) or approximate similarity measures provide efficient methods of determining whether pairs of points are close or far in some metric space. Given this property, LSH mechanisms and other tools underlie multiple algorithms and data structures for processing massive collections of data. Most notably, LSH is a useful technique for data-independent approaches to the Approximate Nearest Neighbor Search problem (ANNS), where the aim of an algorithmic solution is to efficiently find a c-approximate near neighbor such that the fetched point is at most within distance cr of the query point for some constant c with high probability. Other approaches that create data structures allowing for easy queries for similar points can similarly be applied to ANNS as well as other applied tasks in different computational domains. While numerous approaches exist for these tools in Euclidian and Hamming spaces, the problem is made more difficult for alternative spaces and distance measures. The data format and distance measures that this review focuses on are strings and the edit distance, where devising time-efficient approximations has been a long investigated research direction [6, 11] with near optimal sublinear-times only recently [2, 9].

Data in many sub-fields of machine learning and data science rely on strings and other sequentially ordered data. Namely, Natural Language Processing (NLP) is concerned with manipulating and processing textual strings of characters and words, computational genomics is concerned with analyzing collections of gene or peptide sequences, and other fields like computational finance may deal with sequences of categorical data. Algorithms such as LSH are widely useful for de-duplicating and other pre-processing of corpora, though LSH, related approaches, and notions of distance must incorporate both the presence and order of sequences for meaningful measurements in these domains. The most natural similarity or difference measure for sequences is the *edit distance*, measuring the minimum number of edits (a combination of inserting, deleting, or substituting characters or sequence items) required to transform one of the sequences of a pair into the other. This notion is problematic for creating efficient Nearest Neighbor algorithms, as while computing exact Euclidian norm distances requires time linear with the dimension, $O(n)$, exact edit distance computations require time $O(nm)$ for sequences of length n and m [16].

In this review paper, we focus on recent progress for distinguishing close and far edit distances on pairs of strings, including some approximations of edit distance if particularly applicable to efficiently querying similar strings. In particular, we discuss algorithms that form LSH families for the edit distance, solutions to forms of the Gap Edit Distance problem, and string similarity tests for both Levenshtein and Ulam Distance, which are commonly used throughout machine learning sub-fields. The paper is organized as follows: section 2 introduces some of the common notation and important background; section 3 summarizes each approach reviewed including algorithm analyses among three categories (set similarity-based, metric embedding-based, and algorithms that directly operate on strings); and section 4 briefly discusses the application of the algorithms to NLP and computational genomics before concluding with a summary and mention of future work possibilities.

2. Preliminaries and Notation.

2.1. Strings and Sequences.

. As they relate to edit distance and the algorithms covered in this report, we will briefly introduce important properties of strings. First, strings will primarily be denoted by x and y throughout the review, such that for strings of length $d = |x|$, $x = x_1 x_2 ... x_d$ and the same applies to y. Strings are composed of a sequence of character or items drawn from an *alphabet*, which we denote as Σ, such that $x_i \in \Sigma \ \forall i \in [|x|]$. Specific indices and substrings are denoted by x_i and $x_{i...j}$ respectively for some indices $i < j \leq |x|$.

For some string x, it is said to be *non-repeating* if no substring of any length is repeated elsewhere in the string. Additionally, a string is said to be *t-non-repeating* if any substring of length at least t is not repeated elsewhere in the string, relaxing the repetition constraint. For strings that are repetitive, we characterize the repetitiveness of the string through its *period*, or the smallest value p for which $x_i = x_{i+p}\ \forall p \in [0, |x| - p]$.

2.2. Locality Sensitive Hashing. We begin with a brief discussion of Locality Sensitive Hashing [17]. LSH aims to hash two points at close distance to the same value in the hashed space with high probability while also ensuring far points are hashed to different values with high probability. These two probability bounds, P_1 and P_2 respectively, are a primary method of comparing LSH schemes, which are defined below:

For some hash function in an LSH family $h \in \mathcal{F}$, distance function $d(x, y)$, distance threshold r and constant factor c, two probabilities are of interest.

$$\text{If } d(x,y) \leq r, \text{ then } h(x) = h(y)\ w.p. \geq P_1$$
$$\text{If } d(x,y) \geq cr, \text{ then } h(x) = h(y)\ w.p. \leq P_2$$

Desirable LSH families satisfy $P_1 > P_2$, ensuring close points are more likely to be hashed to the same value than far points. So, LSH families can be compared by a measure of quality $\rho = \frac{\log 1/P_1}{\log 1/P_2}$, where larger values of ρ are more desirable.

2.3. Edit Distance. Applying LSH to sequences and strings requires a distance function that incorporate both the overlap and similar ordering of items in a pair of sequences. Distances between sequences are commonly measured through the *edit distance*, broadly signifying the number of edit required to transform one string into another. In this report, we focus on two similar metrics for the edit distance formally defined below: *Levenshtein Distance*, and *Ulam Distance*.

Levenshtein Distance. Given two strings x and y, Levenshtein Distance counts the minimum number of character insertions, deletions, and substitutions necessary to transform x into y, normalized by the length of the larger sequence [19]. Insertion of a new character, e, into string x at index i yields an edited string $x' = x_{1\ldots i}\ e\ x_{i+1\ldots n}$. Deletion of a character at index i yields an edited string $x' = x_{1\ldots i-1}\ x_{i+1\ldots n}$. Finally, a substitution of character x_i with character e yields $x' = x_{1\ldots i-1}\ e\ x_{i+1\ldots n}$. Throughout the report, we denote the Levenshtein distance between x and y as $\mathrm{Lev}(x, y)$. Note the following pertinent properties of Levenshtein distance:

Property 1. $0 \leq |\ |x| - |y|\ | \leq \mathrm{Lev}(x, y) \leq \max\{|x|, |y|\}$ for any strings x, y

Property 2. Let $i \in [|x|]$, $j \in [|y|]$, be values such that with the distance-minimizing set of edits, x_i is edited to y_j. Then

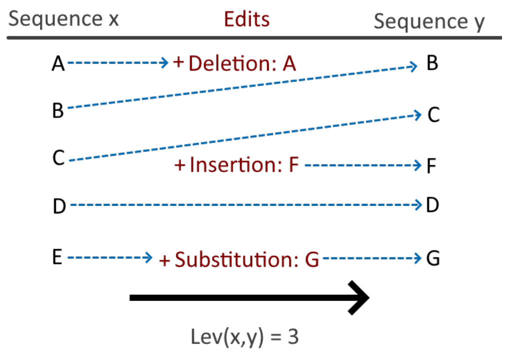

FIG. 1. *Example of applying edits to a string x to transform x into string y, counting the number of edits to compute the Levenshtein distance.*

$$\mathrm{Lev}(x, y) = \mathrm{Lev}(x_{1\ldots i}, y_{1\ldots j}) + \mathrm{Lev}(x_{i+1\ldots|x|}, y_{j+1\ldots|y|})$$

Ulam Distance. Similarly to Levenshtein Distance, Ulam Distance also considers a finite-sized alphabet, Σ, and two strings x and y. However, distinct from Levenshtein distance, Ulam distance only measures the minimum number of insertions and deletions required to transform one string into another, disallowing substitution edits [1]. Additionally, Ulam distance is restricted to non-repetitive strings such that for all characters in x, $x_i \neq x_j \; \forall j \neq i \in n$. We use $\text{Ul}(x, y)$ to refer to the Ulam Distance between strings. Notably, the Ulam distance can be equated to the edit distance using a larger alphabet such that no subsequences are repeated in either string. Additionally, while Ulam distance is primarily intended for non-repetitive strings, or *permutations*, we also consider the Ulam distance as the insertion-delete distance between strings that are not necessarily non-repetitive. Similar to Levenshtein distance, note the following pertinent property of Ulam distance: $\text{Ul}(x, y) = d - |\text{LCS}(x, y)|$ where LCS denotes the longest common subsequence between strings x and y, and $d = \max\{|x|, |y|\}$.

When the particular edit distance metric is not of concern for a specific problem or algorithm, $\text{ED}(x, y)$ will refer to either metric for simplicity.

2.4. Gap Edit Distance Problem. For sequences, the Gap Edit Distance Problem [7] is closely related to the goals of LSH, and is the primary problem considered by many of the reviewed algorithms in this report. Broadly, the problem is in a way a relaxation and approximation of querying a dataset of sequences to test for those identical to the query (that is, $\text{Lev}(x, y) = \text{Ul}(x, y) = 0$ or approximately). The problem is formally defined as follows.

Consider a dataset of strings, X, and a query string y. For each string $x^{(j)} \in X$, an algorithmic solution should return YES if $\text{ED}(x^{(j)}, y) \leq r$ for some edit distance threshold r and return NO if $\text{ED}(x^{(j)}, y) > f(r)$ for some function of the distance threshold, $f(r)$. No guarantees or restrictions are placed on pairs with edit distance between r and $f(r)$.

Solutions to the Gap Edit Distance problem are usually classified by the function $f(r)$, where for example, *Quadratic* Gap Edit Distance refers to solutions to the problem where $f(r) = O(r^2)$. Algorithms for ANNS with strings are then equivalently solutions to the *Linear* Gap Edit Distance problem where $f(r) = cr$ for some constant c.

This is generalized and denoted as (β, α)-GED(X,Y). Where $\alpha \geq \beta \geq 0$. α is usually some function of β like $\alpha = \beta^c$

2.5. Metric Embeddings. Metric Embeddings facilitate the approximation of more complex distance measures by reducing the problem to computing metrics in spaces where simple algorithms already exist. Metric Embeddings rely on the notion of a *metric space*, which is composed of both set of items or points, M, and a *metric*, d that measures distances between members of M, which is defined as $d(x^j, x^{(k)}) : M \times M \to \mathbb{R}; x^{(j)}, x^{(k)} \in M$. A metric space is then the combination of (M, d) with specific properties and constraints. Notably, $d(x^{(j)}, x^{(j)}) = 0, \forall j \in |M|$ and $d(x^{(j)}, x^{(k)}) > 0, \forall x^{(j)} \neq x^{(k)} \in M$, or in other words, the distance between identical points is always 0 and the distance between any distinct points is always positive. Natural choices of metric spaces for points in d-dimensional Euclidian space are $(\mathbb{R}^d, \ell_p)$ where $\ell_p(x, y) = \|x - y\|_p$ defines the p-norm of the difference between points.

Definition 2.1 ((M, d)-Metric Space). *Let an (M, d)-Metric Space be composed of both a set of points or items M and a distance function or metric $d(x^j, x^{(k)}) : M \times M \to \mathbb{R}$ with the following properties.*

1. $d(x^{(j)}, x^{(j)}) = 0, \; \forall j \in |M|$
2. $d(x^{(j)}, x^{(k)}) > 0, \; \forall x^{(j)} \neq x^{(k)} \in M$
3. $d(x^{(j)}, x^{(k)}) = d(x^{(k)}, x^{(j)}), \; \forall x^{(j)}, x^{(k)} \in M$

Using this definition of a metric space, metric embeddings can then be described as follows. For two metric spaces where one is a source and the other is a target space, (M_s, d_s) and (M_t, d_t) respectively, an embedding is a functional mapping $f_e : M_1 \to M_2$ that attempts to conserve some properties of d_s under d_t. Most often, the aim is for $d_s(x^{(j)}, x^{(k)}) = d_t(f(x^{(j)}), f(x^{(k)}))$ or for d_t to approximate d_s with some error so that the mapping can be leveraged for easier calculation of the source metric. When d_t is only an approximation, the effectiveness of the embedding is considered through the *distortion* of the embedding, *distortion*(f), which is defined as follows.

Definition 2.2 (Metric Embedding Distortion). *Let $f : M_s \to M_t$ be the metric embedding from metric space (M_s, d_s) to (M_t, d_t). The measures of expansion(f), contraction(f), and distortion(f) are defined as* *follows.*

$$\begin{aligned}
\textit{expansion}(f) &= \max_{x,y\in M_s} \frac{d_t(f(x), f(y))}{d_s(x,y)} \\
\textit{contraction}(f) &= \max_{x,y\in M_s} \frac{d_s(x,y)}{d_t(f(x), f(y))} \\
\textit{distortion}(f) &= \textit{expansion}(f) \cdot \textit{contraction}(f)
\end{aligned}$$

Therefore, effective metric embeddings seek to construct f such that $\textit{distortion}(f)$ is as small as possible to ensure good approximation of the original distance metric. With this notion of metric embeddings, LSH for sequences can be accomplished by forming an embedding that approximately preserve Levenshtein or Ulam distance when mapping into a space with a defined and effective LSH family.

A related quantity, the Lipshitz norm, calculates the distortion, defined as $\|f\|_{Lip} = \sup_{x\neq y} \frac{d(f(x),f(y))}{d(x,y)}$. In other words, the Lipshitz norm measures the maximum ratio of the distance in the embedded space to the distance in the original space across all pairs of points.

3. Current Approaches. We now direct attention to recent approaches related to distinguishing close and near sequences, LSH, and the Gap Edit Distance problem. Broadly, the approaches are grouped into three categories that highlight the trends in approaches, and focus is given to research that represent distinct perspectives in each group.

3.1. Set Similarity Approximations of Edit Distance. Since computing Levenshtein distances requires pairwise comparisons between whole documents, it was practically infeasible to scale such computations across a large collection of documents. The algorithms outlined below aim to compute a sketch that grows linearly in the size of documents and estimate the Jaccard index between two sketches. By estimating the Jaccard index, we can use the index as a proxy for measuring edit distance:

Definition 3.1 (Jaccard similarity coefficient). *Let U be a universal set consisting of subsets A and B. Then the Jaccard index is the ratio between the cardinality of the intersection between the subsets and the cardinality of the union between the subsets:*

$$J(A,B) = \frac{|A\cap B|}{|A\cup B|}$$

Algorithm 1: MinHash. The MinHash algorithm was developed as an LSH scheme to determine the similarity between web pages and remove duplicate results from search [10]. Given the universal set U and subsets $A, B \subset U$, we can define MinHash as follows:

Definition 3.2 (MinHash). *Let U be a universal set and $S \subseteq U$. We define $h : U \to \mathbb{N}$ and $\sigma : U \to U$, where h is a random hash function and σ is a random permutation performed on the set. We define the MinHash value, $h_{min}(S) = argmin(h(\sigma(S)))$. Therefore, the MinHash of S is defined as the element of S with the minimum hash value.*

Theorem 3.3. *Consider sets A and $B \subseteq U$. Then MinHash is a valid LSH for Jaccard similarity:*

$$Pr[h_{\min}(A) = h_{\min}(B)] = J(A,B)$$

Proof. We apply $h_{\min}$ to $A, B \subseteq U$. The MinHash values are equal only if the minimum hash value lies within $A \cap B$ among the elements in $A \cup B$. Hence, the probability equates to the Jaccard index:

$$Pr[h_{\min}(A) = h_{\min}(B)] = \frac{|A\cap B|}{|A\cup B|} = J(A,B)$$

As we have shown that MinHash is a valid LSH, we can get an unbiased estimation of the set similarity by performing multiple random and independent permutations with D functions:

$$\hat{J}(A,B) = \frac{\sum_{d=1}^{D} 1[h_{\min_d}(A) = h_{\min_d}(B)]}{D}$$

While MinHash provides an unbiased estimate of the Jaccard index, the Jaccard distance does not always correspond to the Levenshtein distance. While a low Jaccard similarity does imply a low edit distance, the correspondence does not apply to a high Jaccard similarity. Consider two sequences S_1 and S_2. S_1 consists of k 0's followed by $n-k$ 1's while S_2 consists of $n-k$ 0's followed by k 1's. When breaking down these sequences into sets of k-mers, we can observe that the sets are identical. Hence, despite the edit distance being $\leq 2k/n$, the Jaccard similarity would equate to 1. The example demonstrates that a high Jaccard similarity does not approximate the edit distance sufficiently. MinHash's variants aim to address these issues.

Algorithm 2: Order Min Hash (OMH). Rather than providing an unbiased estimate of the Jaccard index, OMH proposes estimating the *Weighted Jaccard* to more closely approximate the Levenshtein distance [22].

Definition 3.4 (Weighted Jaccard similarity). *Consider weighted sets $A = [A_1, ..., A_n]$ and $B = [B_1, ..., B_n]$, with A_k and B_k as real-valued weights for $k \in \{1, ..., n\}$. We define the weighted Jaccard as follows:*

$$J^w(A,B) = \frac{\sum_k \min(A_k, B_k)}{\sum_k \max(A_k, B_k)}$$

While the original MinHash algorithm would treat each document as a bag of unordered k-mers, OMH would consider a set of k-mers and their occurrence number. For example, a string S of length n would correspond to the set $M_k^w(S)$. If there are x copies of a certain k-mer m in S, then the set $M_k^w(S)$ would consist of pairs $(m,0), ..., (m, x-1)$. M can be denoted as the 'weighted multi-set' of k-mers where the number of occurrences of the k-mer correspond to the weights. Hence, OMH's estimation of the weighted Jaccard takes the number of occurrences of each element into consideration.

We can define the hash function $h_{l,\pi}^w$ as follows. $h_{l,\pi}^w(S)$ outputs a vector of length l consisting of pairs (m_i, o_i) that are the smallest elements of $M_k^w(S)$ according to permutation π. The pairs are listed in the order in which they appear in the sequence.

Theorem 3.5. *Consider sets A and $B \subseteq U$. When $l = 1$, OMH is an LSH for the weighted Jaccard:*

$$Pr[h_{1,\pi}(A) = h_{1,\pi}(B)] = J^w(A,B)$$

Proof. The proof largely mirrors that of the previous theorem for MinHash. Since every k-mer in the set $M_k^w(A) \cup M_k^w(B)$ has the same probability of being chosen, the probability of collision is equivalent to selecting a k-mer from the intersection $A \cap B$. The probability of picking the k-mer within the intersection is then weighted by the maximum occurrence number. Hence, the probability equates to the weighted Jaccard.

3.2. Embedding Edit Distance in Alternate Spaces. Metric embeddings enable existing algorithms for common spaces like Euclidian and Hamming spaces to approximate similar goals for inputs with more complex structure. So, effective LSH schemes can be formed by embedding the edit distance in these spaces and applying existing, efficient LSH families. So, the effectiveness of existing metric embeddings for the edit distance can be compared using the distortion of the embeddings and the effectiveness of LSH families for the embedded space.

Algorithm 1: Embedding Levenshtein Distance in ℓ_1. Ostrovsky & Rabani [25] formulated a metric embedding from binary strings to ℓ_1 with a low distortion for bit strings. The algorithm generally involves breaking down a string into *blocks* of equal length, and creating a set of overlapping substrings for each block using a *shift* operation:

$$\mathit{shift}(x,s) = \{x_{1...(|x|-s+1)}, x_{1...(|x|-s+1)}, ..., x_{s...|x|}\}$$

The operation effectively passes a sliding window over the input string and extracts all substrings of length s. Using the set of blocks and their associate substrings as a representation of the string, the algorithm then defines the distance between two strings as the truncated ℓ_1 distance between the embeddings, where "truncated" maps any distances. Note that concatenating all the embeddings for all possible values of s for each block entirely preserves the edit distance in ℓ_1, but as an approximation, the truncated ℓ_1 is used on a subset of values of s. Overall, the algorithm posits the following:

THEOREM 3.6. *There exists a constant $c > 0$ such that $\forall d \in \mathbb{N}$ there exists an embedding $\psi : (\{0,1\}^d, ED) \to \ell_1$ with distortion$(\phi) \leq 2^{c\sqrt{\log d \log\log d}}$.*

A sketch and explanation of the central proof is outlined below, beginning with a lemma bounding the embedding approximation. Take s to be the length of each substring generated by *shift*, d to be the (maximum) length of all sequences, and t to be a parameter determining block size, $b = \frac{d \cdot \ln(s/\epsilon}{t}$. Also let $\binom{S}{k}$ denote selecting k items or characters from set or string S. Note that substrings in this context do not necessarily need be consecutive characters in the original string.

LEMMA 3.7. *For an $\epsilon > 0$, and $d, s, t \in \mathbb{N}$ s.t. $\ln(s/\epsilon) \leq t \leq d$, there is an embedding $\psi : \binom{\{0,1\}^d}{s} \to \ell_1$ such that for all inputs $A, B \in \binom{\{0,1\}^d}{s}$*

$$\|\psi(A) - \psi(B)\|_1 \leq \frac{1}{s} \cdot \min_{\sigma}\left\{\sum_{x \in A} \min\{t, 2\mathcal{H}(x, \sigma(x)) \ln(s/\epsilon)\}\right\}$$

and if $\forall x \in A, y \in B, \mathcal{H}(x,y) \geq t$,

$$\|\psi(A) - \psi(B)\|_1 \geq (1-\epsilon)t$$

Where σ represents a bijective mapping from A to B. To select a ψ that satisfies the above, Ostrovsky defines a new function χ. With $A = \{x^1, x^2, ..., x^s\}$ determining a set of d-length strings with cardinality s, $A \in \binom{\{0,1\}^d}{s}$, $\chi(A)$ creates a vector indexed by the set of all b-length strings $z \in \{0,1\}^b$ and all b-length sequences $I = (i_1, i_2, ..., i_b) \in [d]^b$ with ones in indices where $z = x^j_{i_1} x^j_{i_2} ... x^j_{i_3}$ for some j. That is, there is a 1 for every string if it is permutation of some $x \in A$ ordered by a sequence I. With this mapping, $\chi(A)$ creates an embedding $\chi : \binom{\{0,1\}^d}{s} \to \{0,1\}^{(2d)^b}$, and $\psi = \frac{t}{2sd^b}\chi(A)$.

Then, for some $x \in A$, consider the edited sequence $\sigma(x) \in B$ and a randomly selected sequence I. The probability that $\chi(B)_{I,z} = 0$ is bounded by 1 minus the probability that all indices are equal, $x_j = \sigma(x)_j \forall j \in [b]$, which can be defined through the hamming distance between the two strings. So, $Pr[\chi(B)_{I,z} = 0] \leq 1 - (1 - \frac{\mathcal{H}(x,\sigma(x))}{d})^b$. Then using the approximation $1 - x \geq e^{-x}$, we have

$$\begin{aligned} Pr[\chi(B)_{I,z} = 0] &\leq 1 - (1 - \frac{\mathcal{H}(x,\sigma(x))}{d})^b \\ &= 1 - e^{-\frac{2b\mathcal{H}(x,\sigma(x)}{d}} \\ &\leq 1 - e^{-\frac{2\mathcal{H}(x,\sigma(x)\ln(s/\epsilon}{t}} \\ &\leq \min(1, \frac{2\mathcal{H}(x,\sigma(x)\ln(s/\epsilon}{t}) = \frac{1}{t}\min\{t, 2\mathcal{H}(x,\sigma(x)\ln(s/\epsilon)\} \end{aligned}$$

With this, the distance between embeddings is bounded by

$$\begin{aligned}\|\psi(A)-\psi(B)\|_1 &= \frac{t}{2sd^b}\sum_{I\in[d]^b}\sum_{z\in\{0,1\}^b}|\chi(A)_{I,z}-\chi(B)_{I,z}| \\ &\leq \frac{t}{s}\sum_{x\in A}\frac{1}{t}\min\{t,2\mathcal{H}(x,\sigma(x)\ln(s/\epsilon)\} \\ &= \frac{1}{s}\sum_{x\in A}\frac{1}{t}\min\{t,2\mathcal{H}(x,\sigma(x)\ln(s/\epsilon)\}\end{aligned}$$

And for the case that $\mathcal{H}(x,\sigma(x))\geq t\ \forall x\in A$, the same process can be used to find $\|\psi(A)-\psi(B)\|_1\geq(1-\epsilon)t$, which satisfies both original goals. Using these inequalities, the proof then goes on to show that $\|\psi_d\|_{Lip}=2^{O(\sqrt{\log d\log\log d})}$ by induction over sequences of length less than d. Overall, the proof leverages approximations of the edit distance between subsequences of two strings and bounds the distortion through the Lipshitz norm. Notably, the embedding scheme arbitrarily extends to non-binary strings with larger alphabets such as in natural language or gene sequences.

Algorithm 2: Embedding Ulam Distance in ℓ_1. By observation of Ostrovsky and Rabani's result, their approach also achieves *distortion*$(\psi)=2^{O(\sqrt{\log d\log\log d})}$ for the Ulam distance, as it is possible to make it equivalent to Levenshtein distance through data modification. However, **Charikar & Krauthgamer** [13] improve on this distortion for the Ulam metric, leveraging the assumption that no subsequences are repeated and the ability to neglect substitutions. Their approach embedding the edit distance into ℓ_1 achieves a distortion of only $O(\log(d))$:

THEOREM 3.8. *For sequences of maximum length d, there exists an embedding of the Ulam distance into ℓ_1 with distortion $O(\log d)$ and strings into a $\mathbb{N}^{O(|\Sigma|^2)}$ space.*

The resulting embedding takes the form as follows. The embedding $f(P)$ is indexed by all possible coordinates $\{a,b\}$ for $a\neq b\in\Sigma$. Each value is defined as

$$f(x)_{\{a,b\}}=\begin{cases}\frac{1}{x^-1(b)-x^-1(a)} & \text{if } a,b\in x, a<b\\ 0 & \text{otherwise}\end{cases}$$

where $x^-1(a)$ denotes the index of a in string x. The sketch of the proof for Charikar and Krauthgamer's embedding examines the contraction and expansion independently, following the two lemmas below.

LEMMA 3.9. *For strings and non-repetitive permutations x and y,*

$$\|f(x)-f(y)\|_1\leq O(\log d)\cdot Ul(x,y)$$

bounding the expansion of embedding f.

Proof Sketch of Lemma 3.9. Take $H(k)=\sum_{i=1}^{k}\frac{1}{i}$ to be the kth harmonic number. Note that

$$\|f(x)-f(y)\|_1=\sum_{a,b\in\Sigma}|f(x)_{\{a,b\}}-f(y)_{\{a,b\}}|$$

The proof simply observes the case of deleting character x_s in transforming x into y. For such a deletion (or insertion in the case of swapping x and y), each pair of characters $a=x_i,b=x_j$ indexing $f(x)$ contributes $\sum_{j=s+1}^{d}\leq H(d)$ to the total ℓ_1 distance when $i=s$. The case is found to be similar for when $j=s$ and $i<s<j$ as well, both bounded by $H(n)$. Then, it follows that $\|f(x)-f(y)\|_1\leq 3H(d)$. Finally, as harmonic numbers approximate natural logarithms $H(k)\leq 1+\ln k$, the final bound is then $\|f(x)-f(y)\|_1\leq 3(1+\ln n)=O(\log d)$, completing the proof.

Lemma 3.10. *For strings and non-repetitive permutations x and y,*

$$\|f(x) - f(y)\|_1 \geq \frac{1}{8} ED(x, y)$$

bounding the contraction of embedding f.

The proof of Lemma 3.10 is much more involved than the previous, but broadly the analysis involves recursively partitioning a string and examining the quantity $LIS(x)$ as it relates to the Ulam distance. In this case, strings are partitioned such that each even index, $2i$, is randomly assigned a partition, and the preceding odd index, $2i - 1$, is placed opposite. Eventually, the contraction is bounded by $\|f(x) - f(y)\| \geq \frac{1}{8}\text{ED}(x, y)$ satisfying the given lemma.

Following from these lemmas, the total distortion is then

$$\begin{aligned} distortion(f) &= expansion(f) \cdot contraction(f) \\ &\leq \frac{1}{8} \log d = O(\log d) \end{aligned}$$

While primarily for the Ulam distance, the authors go onto show that Levenshtein distance for t-non-repeating strings can be embedded into ℓ_1 with distortion $O(t \log d)$.

Algorithm 3: Embedding Levenshtein Distance into Hamming Space. This section will conclude with an algorithm embedding Levenshtein Distance into Hamming Space rather than ℓ_1 as the above algorithms aim to do. Kociumaka & Saha [18] expand on the constant factor approximation of edit distance devised by Chakraborty et al. [11] to formulate a randomized Hamming Space embedding of edit distance with distortion $O(\text{ED}(x, y))$. While Kociumaka and Saha propose multiple algorithms related to the gap edit distance problem, in this section we focus primarily on the embedding approach. The primary theorem underlying the embedding is as follows, drawn from Chakraborty and colleagues approach:

Theorem 3.11. *For all integers $n \geq 1$, there exists an integer $\ell = O(\log n)$ and embedding function $f : \{0, 1\}^\ell \times \{0, 1\}^{3d}$ such that*

$$\frac{1}{2} ED(x, y) \leq \mathcal{H}(f(x, R), f(y, R)) \leq O(ED^2(x, y))$$

for all $x, y \in \{0, 1\}^n$ with probability at least $\frac{2}{3} - e^{-\Omega(n)}$ over uniform random choice $R \in \{0, 1\}^\ell$. The embedding f can be evaluated in linear time.

While not explicitly mentioned, as an embedding the distortion can be observed by $distortion(f) = \frac{1}{2}\text{ED}(x, y) = O(\text{ED}(x, y))$ with the given probability. The algorithm relies on a set of hash function $H = \{h_1, h_2, ..., h_3 n\}, h_i : \{0, 1\} \rightarrow \{0, 1\}$ and an iterative scan of a string. For a single string x, each character is scanned sequentially, and at each iteration, the current character is added to the embedding, and the hash function determines whether to stay at the current index or increment to the next character in the next iteration. Kociumaka and Saha slightly modify this embedding to achieve an embedding with similar distortion in time $O(\frac{n}{p})$. Instead of hash functions, they simulate the result with the following algorithm.

For two strings, $x, y \in \Sigma^{\leq n}$, parameters $k \geq 0, p \geq 2 \ln d, k, p \in \mathbb{N}$, the algorithm scans both strings in order, tracking the index of each string throughout the scan with two counters, c_x and c_y. Also consider two "coins" that introduce randomness into the embedding. First, a biased coin with probability $\frac{2 \ln n}{p}$ determines whether to compare the current characters x_{c_x} and y_{c_y}. If compared and $x_{c_x} \neq y_{c_y}$, then an unbiased coin (probability .5) determines which counter to increment between c_x and c_y, while also incrementing a variable counting the number of mismatches, c_m. If the initial biased choice to compare the characters fails, then both counters are incremented for the next iteration. At the end of the algorithm, when either $c_x = |x|$ or $c_y = |y|$, the algorithm reports YES if $c_m + \max\{|x| - c_x, |y| - c_y\} \leq 1296k^2$.

The proof of this algorithm involves analyzing a random walk $W \in \mathbb{N}$ that either increments 1 or decrements 1 in each iteration with equal probability. Examining how this walk evolves compared to the algorithm, the authors use the random walk to bound the insertion-deletion distance (Ulam Distance), which then also bounds the Levenshtein distance. In this case, they find that $P[c_m + IDD(x, y) \leq 1296k^2] \geq 1 - \frac{12k}{1296k^2} = \frac{2}{3}$, that is, they affirm that the algorithm returns YES with high probability if $\mathrm{ED}(x, y) \leq k$. In the opposite case, the authors bound the number of mismatches that are skipped throughout the run of the algorithm by examining the probability of many missed mismatches. They find the number of mismatches if at most $(p-1)(c_m+1)$ with probability at least $1 - \frac{1}{d}$, leading to the result that the algorithm returns NO if $\mathrm{ED}(x, y) \geq (1296k^2 + 1)p$.

The embedding involves a small modification to the proposed algorithm, ensuring the Hamming Distance in the embedding space corresponds to the counter c_m, applying the same bounds. For a single string, the embedding uses a random vector $S \subset [1...3n]$ where each value is independently included in S with probability $\frac{2 \ln n}{p}$, and $|S|$ hash functions. These two sources of randomness correspond to the biased and unbiased coins respectively as described above, and the Hamming embedding is iteratively created, setting $f(x)_i$ equal to the current character of x in the scan whenever the looping variable $i \in S$. Overall, the Hamming distance directly approximates the previous counter c_m with high probability, forming the embedding. With this scheme, the size of $|S|$ determines the number of iterations, and the size is controlled by parameter p determining how likely each index is included in S, leading to the result that the embedding can be calculated in $O(\frac{n}{p})$ time.

With these embeddings into ℓ_1 and Hamming space, existing LSH techniques can be applied such as Datar and colleagues' [14] p-stable LSH that can be used to find exact near neighbors in $O(\log n)$ time, RLSH which improves the memory requirements [20], Entropy-based LSH which posits $\rho \approx \frac{2.06}{c}$ for large c [26], and MP-RW-LSH also proposed by Wang and colleagues [27] based on Multi-Probe LSH [21]. For the Hamming space embedding, Bawa and colleagues' LSH Forest with modification by Andoni and colleagues [5] achieves $\rho = \frac{1}{\ln 4c}$, and Andoni and Razenshteyn's data-dependent approach, achieves $\rho = \frac{1}{2c-1}$ (proven optimal) [4].

3.3. Algorithms Directly on Strings.

This section details efficient algorithms that have been applied to the related tasks of approximate pattern matching under Hamming distance and Gap Edit Distance (distinguishing edit distances of r and f(r) as described in section 2.4).

Algorithm 1: Linear Index for approximate pattern matching

K-approximate pattern matching is defined as follows: given an integer $k \geq 0$ and a text S[1..n] over a constant-size alphabet Σ, we build an index for S such that for any query pattern P[1..m], we report efficiently all locations in S that match P with less than or equal to k errors. For the task of exact matching (k = 0), suffix trees require $O(n)$ space and have the optimal matching time of $O(m + occ)$, where occ is the number of occurrences of P in S. However, previous approaches have indicated that significantly larger indexes are required for K-approximate pattern matching. Chan et al. attempt to construct a $O(n)$ index for K-approximate pattern matching with efficient time complexity: Given any pattern P[1..m], their index finds all substrings of S matching P within k errors (the k-error matches of P), in $O(m + occ + polylogn)$ time.

The main novelty of Chan et al.'s index [12] is a check-point technique for handling long patterns. Some particular indices of S are marked as check-points at regular intervals, and special indexing is done for suffixes and prefixes of S terminating at these check-points. For a long pattern, its k-error matches in S will contain some check-points, and the special indexing previously performed at the check-points allows for efficient pattern matching.

For short patterns, we can use a brute force method with the suffix tree, a compressed trie of all suffixes of a given string. For $k = 1$, we can iterate through every 1-error modification of the query pattern in the suffix tree, using $O(n)$ space matching time $O(m^2 + occ)$. In the case of constant k errors, one can perform a brute-force search on a one-error index by repeatedly modifying the pattern at k - 1 different positions and searching for one-error matches. This will take $O(m^k)$ time. Hence, a short pattern of length $logn$ can be handled via this brute force method in polylog n time.

Lets say a pattern is long if it has length at least $\beta \in \mathbb{Z}^+$. At each index α of S that is an integer multiple of β, we denote S[α] as a checkpoint.

We observe the following. Let P[1..m] be a long pattern with $m \geq \beta$ and S be some string of length n. For any k-error match S[j_1.. j_2], there exists an integer $\alpha \in \mathbb{Z}^+$, $j_1 \leq \alpha \leq j_2$ such that S[α] is a check-point and $0 \leq \alpha - j_1 \leq \beta - 1$. Furthermore, let $i = \alpha - j_1 + 1$. Then there exist integers $k_1, k_2 \geq 0$, such that

1. S[α..n] has a prefix that matches P[i..m] with k_1 errors
2. S[1..α - 1] has a suffix that matches P[1..i - 1] with k_2 errors
3. $k_1 + k_2 \leq k$

From these observations, we can find all k-error matches of P in S with the following algorithm:

Algorithm 3.1 k-MATCH(P): finds all k-error matches of P in S, where P is long

```
Initialization:
  Let TAIL be the set of suffixes of S beginning at a check-point, i.e., TAIL = {S[$\alpha$..n] | $\alpha$ = $\beta$, 2$\beta$,...}.
  Let HEAD = {S[1..$\alpha$-1] | $\alpha$ = $\beta$, 2$\beta$,...} be the set of prefixes of S ending just before a check-point
For each $i = 1, ..., \beta$, divide P into P[1..i - 1] and P[i..m]. For all possible $k_1, k_2$ such that $k_1 + k_2 \leq k$:
  Step 1: Find all S[a..n] $\in$ TAIL that have a prefix matching P[i..m] with exactly $k_1$ errors. Let $Tail_{i,k_1}$ be the set of these suffixes.
  Step 2: Find all S[1..b] $\in$ HEAD that have a suffix matching P[1..i - 1] with exactly $k_2$ errors. Let $Head_{i,k_2}$ be the set of these prefixes.
  Step 3: For each S[a..n] $\in Tail_{i,k_1}$ and S[1..b] $\in Head_{i,k_2}$ , if a = b + 1, we report a k-error match of P starting at S[a - i + 1].
```

We denote each successful pair of S[a..n] and S[1..b] that satisfies a = b + 1 to be a "connecting pair." To find the k-approx. matches of a pattern P, we want to find $Tail_{i,k_1}$ and $Head_{i,k_2}$ as efficiently as possible for all i, k_1, k_2. To do so, we use a data structure called an l-error tree for $l = 0, ..., k$. The case for $Tail_{i,k_1}$ and $Head_{i,k_2}$ are symmetric, so we focus on efficient storing of TAIL.

Lemma 3.12. *Let Z be a collection of suffixes of a text S[1..n]. For any integer $l \geq 0$, an l-error tree for Z has the following properties.*

1. *The l-error tree is a collection of trees with $O(|Z|3^l log^l n)$ nodes. Each leaf represents a suffix in Z*
2. *The l-error tree occupies $O(|Z|3^l log^l n)$ words.*
3. *For any pattern P [1..m], there exist $O(6^l log^l n)$ nodes in the l-error tree such that each leaf under these nodes represents a distinct suffix in Z that has a prefix matching Q with exactly l errors. It takes $O(6^l log^l nloglogn)$ time to find these nodes*

For each $l = 0, 1, ..., k$, we store an l-error tree for TAIL, calling them T-error-tree$_0$, T-error-tree$_1$, ... , T-error-tree$_k$. We also store a suffix tree for S. By Lemma 3.12.3, which describes properties of the l-error tree, for any k_1, the k_1-error tree has $O(6^{k_1} log^{k_1} n)$ nodes, where each of the leaves under them represent the distinct suffixes in $Tail_{i,k_1}$. These are the "covering nodes" for $Tail_{i,k_1}$.

Lemma 3.13. *If we have built the trees T-error-tree$_0$, T-error-tree$_1$, ... , T-error-tree$_k$. The space required is $O(n + n/\beta * 3^k log^k n)$ words. Then, for any pattern P[1..m], after we have pre-processed P using a suffix tree, we can find, for any i and k_1, the covering nodes for $Tail_{i,k_1}$ in T-error-tree$_{k_1}$ in $O(6^{k_1} log^{k_1} n * loglogn)$ time.*

This is evident as the total space required for the suffix tree (in words) is $O(n)$, and the space required for the l-error trees, $l = 0, ..., k$ is upper bounded by

$$\sum_{l=0}^{k} O(n/\beta * 3^l log^l n) = O(n/\beta * 3^k log^k n)$$

The time complexity for queries on any pattern P is directly evident from Lemma 3.12.3.

Given $Tail_{i,k_1}$ and $Head_{i,k_2}$ for some i, k_1, k_2 from queries to the l-error trees, we determine k-error matches of P by finding all suffixes S[a..n] $\in Tail_{i,k_1}$ and prefixes S[1..b] $\in Head_{i,k_2}$ such that $a = b + 1$. Through similar exploitation of properties of the l-error tree, it turns out that these connecting pairs S[a..n] and S[1..b] can be found efficiently through a tree-cross-product data structure.

LEMMA 3.14. *We can store an $O(k * n/\beta * 3^k log^{k+1} n)$-word data structure for the l-error-trees. Then using this data structure, for any i, k_1, k_2 with $k_1 + k_2 \leq k$, we can find all connecting pairs between $Tail_{i,k_1}$ and $Head_{i,k_2}$ in $O(6^{k_1+k_2} log^{k_1+k_2} n loglogn + occ)$ time. In this, occ is the number of connecting pairs.*

Finally, our algorithm will have space complexity accounting for TAIL, HEAD, and connecting pairs for all i, k_1, k_2.

$$O(n + n/\beta * 3^k log^k n + k * n/\beta * 3^k log^{k+1} n) = O(n + k * n/\beta * 3^k log^{k+1} n)$$

If we let $\beta = k3^k log^{k+1} n$, the space complexity becomes linear $O(n)$.

Algorithm 2: Approximating Gap Edit Distance in sublinear time

The advent of modern computational paradigms and their massive dataset sizes has catalyzed interest in sublinear-time algorithms for the (β, α)-Gap Edit Distance problem defined in 2.4. Much of the progress in this field has been achieved by using adaptive queries: queries that utilize the output of previous queries to inform subsequent queries. Goldenberg et al. [15] sign a non-adaptive algorithm that answers the (k, k^c)-Gap Edit Distance problem detailed in section 2.4 with query complexity $\tilde{O}(\frac{n}{k^{c-1}})$. They prove this bound is optimal up to poly-logarithmic factors.

The notation $\tilde{O}$ means that poly-logarithmic factors in n are ignored.

Observations

1. For any strings of length n, $X, Y \in \Sigma^n$, for all contiguous substrings indexed by $i, j \in [n]$, ED(X[i...j], Y[i...j]) $\leq$ ED(X,Y)
2. For all strings $X_1, X_2, Y_1, Y_2 \in \Sigma^*$, ED($X_1X_2, Y_1Y_2$) $\leq$ ED(X_1, Y_1)+ED(X_2, Y_2)

Goldenberg et al.'s algorithm is a randomized procedure which makes $O(\beta/\alpha * n)$ queries to the two input strings. It is reduction to the gap edit distance problem. They follow and improve upon the algorithmic structure Andoni and Onak's earlier work [3], which essentially partitions the input strings into blocks of equal sizes and calls an oracle to solve the gap edit distance problem on the randomly chosen block pairs. The results of the oracle calls on the smaller blocks determine the output for the entire string. Consider Algorithm 3.2.

Algorithm 3.2 Andoni and Onak: (β, fβ)-GED(X,Y)

Initialization:

Devise an oracle that solves the (β, α)-gap edit distance problem in almost-linear time $\tilde{O}(n)$. Such oracles have been established in previous results.

Step 1: Partition input strings $X, Y \in \Sigma^n$ into $m = \frac{n}{b}$ blocks of length b. The ith contiguous substring of X is denoted as X_i, and correspondingly for Y_i

Step 2: Using sampling frequency ρ, we sample each of the m blocks with probability at least $\rho = \Omega(b/\alpha)$

Step 3: Given our sampled blocks, we try to search for a "NO witness" - any pair of blocks X_i, Y_i such that ED$(X_i, Y_i) \geq \beta$. If so, by Observation 1, this block being a NO witness $\Rightarrow ED(X, Y) \geq \beta \Rightarrow$ ED(X,Y) = NO. We solve ED on the smaller blocks using the $(\beta, f\beta)$-gap edit distance oracle.

Now, consider the correctness of this approach. Clearly if ED(X,Y) $\leq \beta$, none of the oracle calls on blocks X_i, Y_i will return NO, so it is correct.

For the case of ED(X,Y) $> \alpha$. On average, $ED(X_i, Y_i) > \frac{\alpha}{m}$. Consider two extreme scenarios. In one extreme case, the string differences are uniformly distributed across all of the blocks: ED(X_i, Y_i) = ED(X_j, Y_j) $\forall i, j \in [m]$. We want our $(\beta, f\beta)$-GED(X_i, Y_i) algo to return NO if the number of edits in a block is greater than fβ. Hence, We set m such that it satisfies $\frac{\alpha}{m} \geq f\beta$. The other extreme scenario is where the string differences are concentrated in a few blocks, and the remaining blocks have very few or no string differences. For blocks with a significant number of characters differences, ED(X_i, Y_i) is upper bounded by the block length: ED$(X_i, Y_i) \leq$ b. For blocks with few string differences, we can assume that ED$(X_i, Y_i) =$ 0. Of the m blocks, at least α/b of them must have a significant number of string differences by pidgeon-hole principle. Hence, we need to sample with rate at least $\rho = \Omega(b/\alpha)$.

Minimizing ρ is equivalent to maximizing m. Recall the constraint that $\frac{\alpha}{m} \geq f\beta$. Thus, we set $b = \frac{n}{m} = \frac{n*f\beta}{\alpha}$. The query complexity is thus $O(\rho n) = O(\frac{b}{a}n) = O(\frac{n^2\beta}{\alpha^2})$. The runtime is almost-linear in the query complexity.

Goldenberg et al. use a similar approach, but they use blocks of different length. While query complexity is still O(ρn) for each level and there are now O($\log n$) levels, we can reduce the sampling frequency to $\rho = O(\frac{f\beta}{\alpha})$. If ED(X,Y) $\geq \alpha$, we want to return NO. If we have a shorter block length of size O($f\beta$), again by pidgeon-hole we have that at least $\frac{\alpha}{O(f\beta)}$ of the blocks have a significant number of string differences and return NO upon being queried. Hence, our new sampling rate is $\rho = O(\frac{f\beta}{\alpha})$, yielding a better time complexity than if a fixed block size is used.

Algorithm 3.3 Goldenberg et al.: (β, fβ)-GED(X,Y) using multi-level block sizes

```
Initialization:
  1. Devise an oracle that solves the ($\beta$, f$\beta$)-gap edit distance problem in almost-linear time $\tilde{O}(n)$.
  2. Set sampling rate $\rho = \frac{10\phi}{\alpha}$. $\phi \in \mathbb{Z}^+$ is a parameter.
  3. For every level $p \in [0, ...\lceil logn \rceil]$: Partition strings X, Y into $m_p = \lceil \frac{n}{2^p} \rceil$ blocks of length $2^p$. Denote
     $X_{p,i}$ as block i of level p: $X_{p,i} = X[i * 2^p ...(i+1) * 2^p]$. $Y_{p,i} = Y[i * 2^p ...(i+1) * 2^p]$
Loop:
For each level $p \in [\lceil log(\phi) \rceil ... \lfloor log(\rho n) \rfloor]$, perform $\lceil \rho m_p \rceil$ iterations. Note that smaller block sizes $\Rightarrow$ larger
$m_p \Rightarrow$ more samples for a particular level.
  1. At each iteration and corresponding level $p$, choose one of the available blocks $i \in [0, m_p]$ uniformly at
     random.
  2. Use oracle to solve ($\beta, \phi$)-GED($X_{p,i}, Y_{p,i}$).
  3. Return YES iff all oracle calls across all levels return YES.
```

The total number of calls to the (β, ϕ)-GED$(X_{p,i}, Y_{p,i})$ oracle is upper bounded by the following:

$$O(\sum_{p=\lceil log(\phi) \rceil}^{\lfloor log(\rho n) \rfloor} \lceil \rho m_p \rceil) = O(\sum_{p=\lceil log(\phi) \rceil}^{\lfloor log(\rho n) \rfloor} \frac{\rho n}{2^p}) = O(\frac{\rho n}{\phi}) = O(\frac{n}{\alpha})$$

THEOREM 3.15. *There exists a randomized reduction that, given a parameter $\phi \in \mathbb{Z}^+$ and an instance of (β, α)-GED satisfying specific bounds of α, β, ϕ, solves (β, α)-GED using $O(\frac{n}{\alpha})$ calls to an oracle for $(\beta, \phi) - GED$ using substrings of the original string. This reduction errs with probability at most $\frac{1}{e}$.*

The proof of correctness is slightly involved, but it relies on the fact that ED$(X_{i,p}, Y_{i,P}) \leq$ ED(X,Y) to verify that YES outputs are correct, and that if ED$(X_{i,p}, Y_{i,P}) \geq f\beta$, for some level, the NO output is correct.

Algorithm 3: LSH for Similarity Search Rather than the Gap Edit Distance problem, McCauley [23] focuses instead on the related similarity search problem which resembles ANNS. Additionally, McCauley devises the first and only known LSH scheme that does not involve embedding strings or intermediary representations, while still distinguishing between distances that differ by a constant factor.

The high level approach is as follows: given two string, x and y, the algorithm generates a random sequence of edits, and the hashes, $h(x)$ and $h(y)$, are equal only if the edits precisely transform x into y. This notion is directly analogous to how edit distance is computed, removing the need to rely on proxy or embedded measures of distance. However, at first glance the hash function does not seem to offer efficiency benefits over directly computing edit distance, given edits are generated and checked between strings. So, the review primarily discusses the hash family construction.

Similar to Kociumaka and Saha's approach, the hash involves scanning each symbol of the input string and builds the hashed representation iteratively. Given a string x assumed to be terminated with a \$ symbol (arbitrary symbol to pad the end of the string), let the hash function begin at x_1 and let i represent the current index in the string. The function is parameterized by variable $p \in [0,1]$ which in turn defines two other variables controlling collision probabilities such that $p_a = \sqrt{\frac{p}{1+p}}$ and $p_r = \frac{\sqrt{p}}{\sqrt{1+p}-\sqrt{p}}$. The hash function relies on a random function $\rho_h : \Sigma \cup \{\$\} \times [1, \frac{8d}{1-p_a} + 6\log n] \to [0,1) \times [0,1)$, that is, it maps from a character in the alphabet and the hash position to a pair of real-valued coordinates < 1. Finally, s stores the hash value as it is constructed such that for each s_i, $s_i \in \Sigma$. s is initialized to an empty string, and the scan begins at index $i = 0$.

As the hash function scans the input string, it first computes the coordinate pair $r = (r_1, r_2) = \rho(x_i, |s|)$ for the current index i. There are three possible events based on the output of ρ.

$$\begin{cases} \textbf{hash-insert: } r_1 \le p_a & \text{append } \perp \text{ to } s \\ \textbf{hash-replace: } r_1 > p_a \ \& \ r_2 \le p_r & \text{append } \perp \text{ to } s \text{ and increment } i \\ \textbf{hash-match: } r_1 > p_a \ \& \ r_2 > p_r & \text{append } x_i \text{ to } s \text{ and increment } i \end{cases}$$

and the scan continues until $i > |x|$ or s breaches the maximum hash value length, $\frac{8d}{1-p_a} + 6\log n$.

For this hash function, the main result characterizing the LSH effectiveness is posited in the following lemmas. Let n be the number of strings in the input dataset.

Lemma 3.16. *For two strings, x and y, that satisfy ED$(x,y) \le r$,*

$$P_\rho[h_\rho(x) = h_\rho(y)] \ge p^r - \frac{2}{n^2}$$

Lemma 3.17. *For two strings, x and y, that satisfy ED$(x,y) \ge cr$,*

$$P_\rho[h_\rho(x) = h_\rho(y)] \le (3p)^{cr}$$

Therefore, the overall sensitivity is characterized by $\rho = \frac{\log 1/(p^r - \frac{2}{n^2})}{\log 1/(3p)^{cr}} \approx O(\frac{r}{cr}) = O(\frac{1}{c})$ for large n.

Proof sketch of lemma 3.16. The proof of the lower bound on the collision probability for close points first notes that if ED$(x,y) \le r$, then there must exist a sequence of transformations or edits, T, that transforms x into y such that $|T| = r$. The proof views the iterative application of the hash operations **hash-insert, hash-replace, and hash-match** as a grid walk over a structure similar to a finite state machine such that at each coordinate (i,j) where i indexes x and j indexes y, the walk can perform any of the three operations or stop. Notably, the unraveled directed graph that the random walk is conducted on resembles the table used in the dynamic programming solution to exactly compute ED(x,y). Additionally, arcs to other edit operations are only defined for coordinates such that $x_i \neq y_i$.

The proof then assumes a transformation of length t that transforms x into y, and bounds the probability that the corresponding random walk is valid in the graph, which then implies that x and y hash to the same value. Because some transitions through the walk will correspond to matching characters, T can divided into a sequence where no edits are made and the sequence of hash operations on x. By the definitions of the hash operations, the probability that the sequence of editing operations, g', exists as a walk in the graph, G can be represented by $P[g' \in G] = p^t$. Given that walk decisions outside of edits (i.e. applying no operation) are guaranteed to exist by the construction of G, the combined probability can also be shown to be p^t. Through a much more involved calculation, the authors show that the probability that the walk *completes* within the maximum number of iterations, which is also the maximum length of the hash value $\frac{8d}{1-p_a} + 6\log n$, is $\geq 1 - \frac{2}{n^2}$. With these two probabilities, the overall bound is

$$P_{h\in\mathcal{H}}[h(x) = h(y)] \geq p^t - \frac{2}{n^2} \tag{3.1}$$

for strings such that ED$x, y \leq r$.

Proof sketch of lemma 3.17. For the upper bound on the collision probability for far points, the proof first notes that $P_{h\in\mathcal{H}}[h(x) = h(y)] = \sum_{T\in\mathcal{T}} p^{|T|}$, where $\mathcal{T}$ denotes the set of valid transformations. Alternatively to the previous view as a directed graph, the problem is now examined as a trie where each node represents a possible intermediary $T_i \in \mathcal{T}$ such that it branches into a maximum of three other nodes (corresponding to incrementing i, j, or neither in the random walk). Then, consider a node, T_i, past a maximum depth $i > cr$. In collapsing a set of siblings T_i, T_{i+1}, T_{i+2} (or a subset that exists), the total probability across all transformations $\mathcal{T}$ is increased by at least $p^{i-1} - 3p^i$ as one transformation sequence of length $i-1$ (i.e. the parent node) is now valid and three transformation sequences of length i (i.e. the set of siblings) were removed. As this process is repeated for all $T \in \mathcal{T}_\rangle$ *s.t.* $depth(T_i) = cr$, at most 3^{cr}, the final set of transformations is of size at most 3^{cr} each of which have probability p^{cr}. Therefore, the final collision probability is bounded by

$$P_{h\in\mathcal{H}}[h(x) = h(y)] \leq (3p)^{cr}$$

for strings such that ED$(x, y) \geq cr$.

As the full approach is a proposed solution to ANNS for strings, the authors go onto show that the generating the random underlying functions ρ for $R = \theta(\frac{1}{P_1})$ independent hash functions, the total space needed to store the LSH mechanism is $O(3^r n^{1+1/c} + dn)$. Additionally, constructing the hashes across the dataset requires time $O(d3^r n^{1/c})$.

4. Discussion and Conclusion.

While strings are the primary data form of concern for numerous lines of research in NLP and computational genomics, the two domains have stark differences that can effect decisions related to what algorithms to use. For example, a stark difference lies in the normal vocabulary or alphabet sizes. NLP data can have vocabularies ranging from 26 letters for character-based tasks to a massive collection of full words, with the latter being most common in modern tasks. Contrarily, computational genomics and biology often only work with an alphabet of 4 nucleotides (A,T,C,G) or in other cases protein sequences composed of 20 possible amino acids. With this and other differences in the composition and use of string data, different algorithms offer different benefits to each field.

4.1. Natural Language Processing Applications.

NLP problems such as detecting data leakage between corpora of text or evaluating language model generations with the ROUGE metric [8] among others can greatly benefit from algorithms that can distinguish close and far strings. In particular, Charikar and Krauthgamer's [13] approach promises low distortion embeddings into ℓ_1 provided guarantees concerning t-non-repetitive strings that can be computed efficiently before applying an LSH family such as Multi-Probe LSH, making it a feasible and efficient tool for natural language problems where patterns in strings are fairly distinct. Additionally, McCauley's similarity search approach can also be applied effectively, but large datasets may prohibit it given the greater dependence on dataset size. However, given the lack of repetition and large vocabularies, set similarity measures are largely innapropriate for language data.

4.2. Computational Genomics Applications.

Approximating sequence similarity is central to bioinformatics pipelines. Most downstream genomic analyses are derived from read alignment, the process of aligning reads taken from a sample individual's genome to a known reference genome. The alignment relies on finding reads similar to regions in the reference genome. While the original MinHash uses a bag-of-words approach for comparing documents, which is suitable for NLP applications, the alphabet of genomic sequences consists of only four characters. Hence, the OMH algorithm was specifically designed LSH to develop algorithmic improvements for read alignment by considering the relative order of k-mers in the sequence [22]. Further extensions of MinHash have been developed for read alignment. In particular, Mash derives a p-value significance test and uses a new distance metric which estimates the mutation rates between genomic sequences [24].

4.3. Conclusion. In the preceding sections, we summarize a selected set of papers revolving around the primary goal of distinguishing strings by an edit distance threshold. Each primarily offers a method of constructing an LSH family for the edit distance, a solution to the Gap Edit Distance problem, or some test of string similarity to classify close from far pairs of strings in edit distance. Within our selection, the approaches are grouped into three broad categories: set similarity proxy measures, metric embeddings into alternative spaces, and algorithms that operate directly on strings

We also discuss the effectiveness of different solutions for data in NLP and computational genomics, two fields that are primarily concerned with sequence data. The highlighted differences between the common forms of data in each field can differentiate the most applicable algorithm to problems in either field, such as alphabet sizes, sequence lengths, and repetitiveness. However, the characteristics of data in both fields, such as the maximum length of sequences that can be reasonably processed by machine learning models, make algorithms that rely on embedding distortions or approximations with large constant factors inapplicable.

These observations can help identify directions for future work and desirable characteristics for algorithms as they apply to these domains. For example, one of the primary obstacles for applying metric embeddings to NLP and genomics data seems to be high constant factors in the distortion terms, which make them inapplicable for shorter sequences. Additionally, taking inspiration from approaches in the last category including pattern matching and LSH operating directly on strings could lead to future algorithms that remove the need for intermediary spaces which worsen distortion and efficiency.

REFERENCES

[1] D. Aldous and P. Diaconis, *Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem*, Bulletin of the American Mathematical Society, 36 (1999), pp. 413–432.

[2] A. Andoni and N. S. Nosatzki, *Edit distance in near-linear time: it's a constant factor*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), 2020, pp. 990–1001, https://doi.org/10.1109/FOCS46700.2020.00096.

[3] A. Andoni and K. Onak, *Approximating edit distance in near-linear time*, in Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, STOC '09, New York, NY, USA, 2009, Association for Computing Machinery, p. 199–204, https://doi.org/10.1145/1536414.1536444, https://doi.org/10.1145/1536414.1536444.

[4] A. Andoni and I. Razenshteyn, *Optimal data-dependent hashing for approximate near neighbors*, 2015, https://arxiv.org/abs/1501.01062.

[5] A. Andoni, I. Razenshteyn, and N. S. Nosatzki, *LSH Forest: Practical Algorithms Made Theoretical*, pp. 67–78, https://doi.org/10.1137/1.9781611974782.5, https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.5, https://arxiv.org/abs/https://epubs.siam.org/doi/pdf/10.1137/1.9781611974782.5.

[6] Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar, *Approximating edit distance efficiently*, in 45th Annual IEEE Symposium on Foundations of Computer Science, 2004, pp. 550–559, https://doi.org/10.1109/FOCS.2004.14.

[7] 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 thirty-fifth annual ACM symposium on Theory of computing, 2003, pp. 316–324.

[8] L. Benkova, *Current metrics for automatic evaluation of machine translation*, DIVAI 2022, (2022).

[9] K. Bringmann, A. Cassis, N. Fischer, and V. Nakos, *Improved sublinear-time edit distance for preprocessed strings*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, https://doi.org/10.4230/LIPICS.ICALP.2022.32, https://drops.dagstuhl.de/opus/volltexte/2022/16373/.

[10] A. Z. Broder, *On the resemblance and containment of documents*, in Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), IEEE, 1997, pp. 21–29.

[11] D. Chakraborty, D. Das, E. Goldenberg, M. Koucký , and M. Saks, *Approximating edit distance within constant factor in truly sub-quadratic time*, Journal of the ACM, 67 (2020), pp. 1–22, https://doi.org/10.1145/3422823, https://doi.org/10.1145%2F3422823.

[12] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S. S. Wong, *A linear size index for approximate pattern matching*, vol. 9, 07 2006, pp. 49–59, https://doi.org/10.1007/11780441_6.

[13] M. Charikar and R. Krauthgamer, *Embedding the ulam metric into l1*, Theory of Computing, 2 (2006), pp. 207–224.

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, *Locality-sensitive hashing scheme based on p-stable distributions*, in Proceedings of the twentieth annual symposium on Computational geometry, 2004, pp. 253–262.

[15] E. Goldenberg, T. Kociumaka, R. Krauthgamer, and B. Saha, *Gap edit distance via non-adaptive queries: Simple and optimal*, 2022, https://arxiv.org/abs/2111.12706.

[16] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997, https://doi.org/10.1017/CBO9780511574931.

[17] P. Indyk and R. Motwani, *Approximate nearest neighbors: Towards removing the curse of dimensionality*, in Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98, New York, NY, USA, 1998, Association for Computing Machinery, p. 604–613, https://doi.org/10.1145/276698.276876, https://doi.org/10.1145/276698.276876.

[18] T. Kociumaka and B. Saha, *Sublinear-time algorithms for computing & embedding gap edit distance*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2020, pp. 1168–1179.

[19] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, in Soviet physics doklady, vol. 10, 1966, p. 707.

[20] Y.-H. Lu, T.-H. Ma, S.-M. Zhong, J. Cao, X. Wang, and A. Al-Dhelaan, *Improved locality-sensitive hashing method for the approximate nearest neighbor problem*, Chinese Physics B, 23 (2014), p. 080203.

[21] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, *Multi-probe lsh: efficient indexing for high-dimensional similarity search*, in Proceedings of the 33rd international conference on Very large data bases, 2007, pp. 950–961.

[22] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford, *Locality-sensitive hashing for the edit distance*, Bioinformatics, 35 (2019), pp. i127–i135.

[23] S. McCauley, *Approximate similarity search under edit distance using locality-sensitive hashing*, CoRR, abs/1907.01600 (2019), http://arxiv.org/abs/1907.01600, https://arxiv.org/abs/1907.01600.

[24] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, *Mash: fast genome and metagenome distance estimation using minhash*, Genome biology, 17 (2016), pp. 1–14.

[25] R. Ostrovsky and Y. Rabani, *Low distortion embeddings for edit distance*, J. ACM, 54 (2007), p. 23–es, https://doi.org/10.1145/1284320.1284322, https://doi.org/10.1145/1284320.1284322.

[26] R. Panigrahy, *Entropy based nearest neighbor search in high dimensions*, in Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, SODA '06, USA, 2006, Society for Industrial and Applied Mathematics, p. 1186–1195.

[27] H. Wang, J. Meng, L. Gong, J. Xu, and M. Ogihara, *Mp-rw-lsh: An efficient multi-probe lsh solution to anns-l1*, Proc. VLDB Endow., 14 (2021), p. 3267–3280, https://doi.org/10.14778/3484224.3484226, https://doi.org/10.14778/3484224.3484226.

Appendix A. Summary of Notation.

Symbol	Description
Σ	Alphabet of symbols or characters to form strings
x, y	A pair of sequences or strings with alphabet Σ
$\mathrm{ED}(x, y)$	Unspecified edit distance between a pair of strings, x and y, referring either to Levenshtein or Ulam Distance
$\mathrm{Lev}(x, y)$	Levenshtein Distance between a pair of strings, x and y
$\mathrm{Ul}(x, y)$	Ulam Distance between a pair of strings, x and y
$\mathrm{LCS}(x, y)$	Longest Common Subsequence between a pair of strings, x and y
(M, d)	Metric space composed of a set of points or items, M and distance metric d
f	Metric embedding from some source metric space to a target metric space
$distortion(f)$	The distortion of a metric embedding, f
$contraction(f)$	The contraction of a metric embedding, f
$expansion(f)$	The expansion of a metric embedding, f
$\mathcal{H}(x, y)$	The Hamming distance between points x and y
$\|f\|_{Lip}$	The Lipshitz (semi)norm for a metric embedding f
$H(k)$	The kth harmonic number, i.e. $H(k) = \sum_{i=1}^{k} \frac{1}{i}$

TABLE 1
Table of the more common notation used throughout the review.