Almost Linear Size Edit Distance Sketch

Michal Koucký*
koucky@iuuk.mff.cuni.cz
Computer Science Institute of Charles University,
Charles University
Prague, Czech Republic
DIMACS, Rutgers University
Piscataway, New Jersey, USA

Michael E. Saks
msaks30@gmail.com
Department of Mathematics, Rutgers University
Piscataway, New Jersey, USA

ABSTRACT

We design an almost linear-size sketching scheme for computing edit distance up to a given threshold k. The scheme consists of two algorithms, a sketching algorithm and a recovery algorithm. The sketching algorithm depends on the parameter k and takes as input a string x and a public random string ρ and computes a sketch $sk_\rho(x;k)$, which is a compressed version of x. The recovery algorithm is given two sketches $sk_\rho(x;k)$ and $sk_\rho(y;k)$ as well as the public random string ρ used to create the two sketches, and (with high probability) if the edit distance $\mathbf{ED}(x,y)$ between x and y is at most k, will output $\mathbf{ED}(x,y)$ together with an optimal sequence of edit operations that transforms x to y, and if $\mathbf{ED}(x,y) > k$ will output LARGE. The size of the sketch output by the sketching algorithm on input x is $k2^{O(\sqrt{\log(n)\log\log(n)})}$ (where n is an upper bound on length of x). The sketching and recovery algorithms both run in time polynomial in n. The dependence of sketch size on k is information theoretically optimal and improves over the quadratic dependence on k in schemes of Kociumaka, Porat and Starikovskaya (FOCS'2021), and Bhattacharya and Koucký (STOC'2023).

CCS CONCEPTS

• **Theory of computation → Sketching and sampling**.

KEYWORDS

Edit distance, sketching, hierarchical Hamming sketch

ACM Reference Format:
Michal Koucký and Michael E. Saks. 2024. Almost Linear Size Edit Distance Sketch. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24), June 24–28, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3618260.3649783

*Part of the work was carried out during an extended visit to DIMACS, with support from the National Science Foundation under grant number CCF-1836666 and from The Thomas C. and Marie M. Murray Distinguished Visiting Professorship in the Field of Computer Science at Rutgers University. Partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 19-27871X and 24-10306S. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 823748 (H2020-MSCA-RISE project CoSP).

STOC '24, June 24–28, 2024, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0383-6/24/06
https://doi.org/10.1145/3618260.3649783

1 INTRODUCTION

The *edit distance* of two strings x and y measures how many edit operations (removing a symbol, inserting a symbol or substituting a symbol by another) are needed to transform x to y. Computing edit distance is a classical algorithmic problem. For input strings of length at most n, edit distance can be computed in time $O(n^2)$ using dynamic programming [13, 19, 22]. Assuming the Strong Exponential Time Hypothesis (SETH), this cannot be improved to truly sub-quadratic time $O(n^{2-\epsilon})$ [2]. When parameterized by the edit distance $k = \mathbf{ED}(x,y)$, the running time has been improved to $O(n+k^2)$ [18]. The edit distance of two strings can be approximated within a constant factor in time $O(n^{1+\epsilon})$ [1, 5, 6, 17].

This paper concerns *sketching schemes* for edit distance, which consist of a *sketching algorithm*, parameterized by an integer k, that takes a string x and (using a public random string ρ) maps it to a short *sketch* $\mathrm{SK}_\rho(x;k)$, and a *recovery algorithm* that takes as input two sketches $\mathrm{SK}_\rho(x;k)$ and $\mathrm{SK}_\rho(y;k)$ and the public random string ρ and, with high probability (with respect to ρ), outputs $\mathbf{ED}(x,y)$ when $\mathbf{ED}(x,y) \le k$ and outputs LARGE otherwise.

The goal is to get polynomial time sketch and recovery algorithms that achieve the smallest possible sketch length. Jin, Nelson and Wu [14] proved that sketches must have length $\Omega(k)$. For edit distance it is not apriori clear whether sketches of size $k^{O(1)}n^{o(1)}$ exist, even non-uniformly. The first sketching scheme with $poly(k)$ sketch size was found by Belazzougui and Zhang [3] who attained sketch size $\widetilde{O}(k^8)$. (Here, and throughout the paper, $\widetilde{O}(t)$ means $t\log^{O(1)} n$, where n is a an upper bound on the length of the strings.) This was improved to $\widetilde{O}(k^3)$ by Jin, Nelson and Wu [14], and then to $\widetilde{O}(k^2)$ by Kociumaka, Porat and Starikovskaya [16]. The above sketches used *CGK random walks* on strings [7] to embed the edit distance metric into the Hamming distance metric with distortion $O(k)$, plus additional techniques. A different approach, based on a string decomposition technique, was used by Bhattacharya and Koucký [4]. Here we will also use this technique.

The quadratic dependence on k in the (very different) sketches of [16] and [4] and also in the exact computation algorithm of [18] is suggestive that there may be something intrinsic to the problem that requires quadratic dependence on k for both sketching and evaluation. In this paper, we show that this is not the case, by presenting an efficiently computable sketch of size $O(k2^{O(\sqrt{\log(n)\log\log(n)})})$ which is k times a "slowly" growing function of n, that is intermediate between $\log^{\omega(1)} n$ and $n^{o(1)}$. Our main result is:

THEOREM 1.1 (SKETCH FOR EDIT DISTANCE). *There is a randomized sketching algorithm ED-SKETCH that on an input string x of length at most n with parameter $k < n$ and using a public random string*

ρ produces a sketch $sk_\rho(x)$ of size $O(k2^{O(\sqrt{\log(n)\log\log(n)})})$, and recovery algorithm ED-RECOVER such that given two sketches $sk_\rho(x)$ and $sk_\rho(y)$ for two strings x and y and ρ, with probability at least $1 - 1/n$ (with respect to ρ), outputs $\boldsymbol{ED}(x, y)$ if $\boldsymbol{ED}(x, y) \le k$ and LARGE otherwise. The running time of the ED-SKETCH is $n^{O(1)}$ and the running time of ED-RECOVER is $\widetilde{O}(\min(n^2, k^3 2^{O(\sqrt{\log(n)\log\log(n)})}))$.

We remark that we did not attempt to optimize the running time, or poly-log factors in the sketch sizes. The running time of ED-SKETCH is largely determined by the running time of the Ostrovsky-Rabani embedding [20] (see below) which runs in polynomial time, but we don't know the exponent. The amount of randomness the algorithm uses can be reduced to poly-logarithmic in n.

Our sketch has the additional property that the recovery algorithm also determines an optimal sequence of edit operations that transforms x to y.

Sketching for Hamming distance has been studied extensively and is well understood. Several approaches yield sketches of size $\widetilde{O}(k)$ that can recover the Hamming distance and can solve the harder problem of *mismatch recovery*, i.e., reconstructing the set of positions where the two strings differ, see e.g. [9, 21]; this sketch size is information theoretically optimal. Our construction for edit distance uses sketches for Hamming distance, but we need a more refined version of Hamming distance sketches that allow for recovery of all differences in regions of the strings where the density of differences is low, even if the overall Hamming distance is large. We call this new sketch a *hierarchical mismatch recovery scheme*. This is the main new technical tool.

Not much is known about sketching edit distance when we only want to approximate edit distance from the sketches. For Hamming distance, there are known sketches of poly-logarithmic size in n that allow recovery of Hamming distance within a $(1+\epsilon)$-factor [10]. For edit distance nothing like that is known. A more stringent notion of sketching is that of embedding edit distance metrics into ℓ_1 metrics. The best known result in this direction, by Ostrovsky and Rabani [20], gives an embedding of edit distance into ℓ_1 with *distortion* (approximation factor) $s_{\mathrm{OR}}(n) = 2^{O(\sqrt{\log(n)\log\log(n)})}$. Interestingly, our sketch relies on this embedding to *fingerprint* strings by their approximate edit distance. The dependence of our sketch size on n in Theorem 1.1 can be stated more precisely as $\widetilde{O}(s_{\mathrm{OR}}(n)^2)$. Since our use of their embedding is "black box", any improvement on the distortion factor for embedding edit distance into ℓ_1 would give a corresponding improvement in our sketch size.

2 OUR TECHNIQUE

The starting point of our sketch is the string decomposition algorithm of Bhattacharya and Koucký [4], The algorithm BASIC-DECOMP (see Section 4.3) takes a string x and partitions it into fragments so that each fragment can be described concisely by a small context-free grammar. The size of the grammar is at most k' for some chosen parameter k'. (In [4] this k' is chosen to be $\widetilde{O}(k)$.) The partitioning uses randomness to select the starting point for each fragment and has the property that for any given position in the string, the probability that it will start a fragment is at most $p \approx 1/k'$.

The partitioning process is *locally consistent*, i.e., when applied to two strings x and y with $\mathbf{ED}(x, y) \le k$ then with probability at least $1 - \widetilde{O}(\mathbf{ED}(x, y)/k')$ the decompositions of x and y are *compatible*, which means that they have the same number of fragments and $\mathbf{ED}(x, y)$ is equal to the sum of edit distance of corresponding fragments. We say that a pair of fragments of x and y is consistent if they are corresponding fragments in some compatible decomposition.

If x and y are compatibly split then to recover $\mathbf{ED}(x, y)$ it suffices to reconstruct each corresponding pair of fragments of x and y that are different; then the edit distance is just the sum of the edit distances of these pairs.

In [4] this decomposition procedure was used to obtain edit distance sketches of size $\widetilde{O}(k^2)$ by reducing the problem of sketching for edit distance to the easier problem of sketching for (Hamming) mismatch recovery mentioned in the introduction. This is the problem of sketching two strings so that from the sketches of two equal length strings x and y, one can recover all the locations where x and y differ. As mentioned earlier, there are known sketches of size $\widetilde{O}(k)$ for this problem.

The reduction to mismatch recovery sketches is as follows. Since each fragment has a grammar of size (number of rules) k', each such grammar can be encoded by a bit string of length $\mathrm{poly}(k')$ so that each encoding has $\widetilde{O}(k')$ 1's, and for two different fragments, (1) the grammar encodings have at most $\widetilde{O}(k)$ mismatches and (2) given the set of locations where the grammar encodings differ one can entirely recover both grammars and therefore both fragments. To sketch a string x we construct the grammar encodings of each fragment in its decomposition, concatenate them, and then compress this using a mismatch recovery sketch. Given the sketches of x and y, the mismatch recovery algorithm finds all the locations where their concatenated grammar encodings differ. From these, one can reconstruct each corresponding pair of unequal fragments, compute the edit distance for each pair, and sum them. If $\mathbf{ED}(x, y) \le k$ then there are at most k corresponding pairs of fragments that differ, and each pair differs in at most $k' = \widetilde{O}(k)$ bits, so the mismatch recovery sketch must handle up to $\widetilde{O}(k^2)$ mismatches, which can be done with sketches of size $\widetilde{O}(k^2)$.

The sketch length is $\widetilde{O}(k^2)$ because the sketch must handle two different extremes. If all edit operations appear in large clusters of size k/C, for $C = \widetilde{O}(1)$, and each cluster is contained in a single fragment pair, then there are at most C fragment pairs that are unequal and these could be handled by sketches of size $\widetilde{O}(k)$. On the other hand, if the edit operations are well separateds, then we could choose a value of k' that is $\widetilde{O}(1)$ resulting in a partition into much smaller fragments each of which has grammar size $\widetilde{O}(1)$. In this case the edit operations may appear in $\Omega(k)$ different fragment pairs, but because the grammar size of the fragments is $\widetilde{O}(1)$ we can again manage with sketch size $\widetilde{O}(k)$. Of course, the distribution of edit operations will rarely match either of these extremes, there may be clusters of edit operations of varying size and density.

Decomposition tree. A natural approach to handling this is to build decompositions for many different values of k'. We start with a decomposition obtained from parameter $k' = k_0$ which is larger, but not much larger than k. We then apply the decomposition to each fragment of the first decomposition, using the smaller parameter $k_1 = k_0/2$. We iterate this recursively where the value k_i of k' at

recursion level i is $k_0/2^i$ where k_0, stopping when k_i is $\widetilde{O}(1)$. This would decompose the string into smaller and smaller fragments ending with fragments described by constant size grammars. We call this a *decomposition tree* of the string. We can then do separate sketches for each of the levels of the tree. The sketches at the top of the tree (small i) are used to find edit operations that occur together in a large cluster, and there can't be too many of these clusters. Sketches at the bottom of the tree (large i) are used to find edit operations in regions where the edit operations are well spread. These edit operations may be spread over $\Omega(k)$ pairs of fragments, but each such pair has edit distance $\widetilde{O}(1)$. The intermediate levels would handle cases where the density of edit operations is intermediate between these extremes.

To be more explicit, the sketch associated to decomposition level i is responsible for recovering pairs of compatible fragments at level i whose edit distance is (roughly) comparable to or larger than k_i, that could not be recovered from the sketches for previous levels of the decomposition. There are at most k/k_i such pairs of fragments (counting only compatibly split pairs). Since the fragments are represented by grammars of size at most k_i, the level i scheme will need to find at most $2k$ mismatches.

For compatible pairs of fragments at level i whose edit distance is small compared to k_i, the decomposition procedure will split them compatibly (with *fairly high probability*[1]) and the edit distance for these will be recovered from the sketches corresponding to deeper levels of the decomposition. In this way, all of the edit operations will be found by some level of the decomposition.

While the sketch for level i only needs to identify at most $2k$ mismatches, it can not use an ordinary mismatch recovery sketch for $2k$ mismatches, because the strings of x and y consisting of the grammar encodings for their level i fragments may differ in many more than $2k$ positions, since the $2k$ positions account only for the differences due to pairs of fragments that must be recovered at level i. but not those due to differences coming from other fragment pairs. To deal with this, we will need the extension of mismatch recovery mentioned earlier that handles hierarchical mismatch recovery (which will be discussed in more detail later in this section.)

Grammar representation. The sketch at a level i encodes the concatenation of the grammars of size at most k_i that represent the fragments in the level i decomposition. We aim to use this sketch to recover those pairs of corresponding fragments whose edit distance is roughly k_i but were not recovered from the sketches at earlier levels. Fragment pairs having edit distance less than k_i will not be recovered by level i but will be further partitioned so that the differences between them will be recovered at deeper levels of the decomposition tree. The grammar representations we use have the important property that for two fragments of edit distance d, the encodings of their grammar are at Hamming distance at most $\widetilde{O}(d)$. This ensures that if two strings x and y are compatibly split, then the hamming distance between the concatenation of the grammar encodings of their fragments will be $\widetilde{O}(\mathbf{ED}(x, y))$. This is important because the size of mismatch recovery sketches is at least the Hamming distance between the strings.

[1]Here *fairly high probability* means probability at least $1-1/\text{poly}\log n$ for a sufficiently large poly-logarithmic function.

Watermarking using edit distance fingerprints. For the level i encoding, we want that if two corresponding fragments are at edit distance at least k_i, then recovering the mismatches in their grammar encodings is enough to fully recover both grammars and therefore both fragments. To ensure this, we will *watermark* the grammar encodings by a special fingerprint (computed from the entire fragment), and replace each 1 in the grammar encoding by the fingerprint, Assuming that the watermark of the two fragments are different, then this will allow one to recover both grammars from the set of mismatches. The watermark we use is a *threshold edit distance fingerprints*. This is a randomized fingerprinting scheme depending on parameter t, that maps each string to an integer so that if two strings have edit distance more than t, their fingerprints differ with high probability, and if two strings have edit distance less than t/P, for some parameter $P > 1$ then their fingerprints will be the same with fairly high probability (with no promise if the edit distance lies in the interval $[t/P, t]$.) The parameter P is a measure of the quality of the fingerprinting algorithm; with smaller P being higher quality. Such a fingerprinting scheme can be obtained from an embedding of the edit distance metric into the ℓ_1 metric. We use the embedding of Ostrovsky and Rabani [20] which has distortion $s_{\text{OR}} = 2^{\sqrt{\log n \log\log n}}$ and this distortion translates into the parameter P of the fingerprinting scheme.

Canonical edit operations. For the sketching procedure as outlined, the recovery algorithm will recover corresponding pairs of unequal but compatible fragments and compute their edit distance. While the recovered fragment pairs are likely to encompass most edit differences between the two strings, it will typically miss some pairs, and may also incorrectly recover a small fraction of the fragment pairs. This means that the set of edit operations recovered will only be approximate. To address this we will need to do multiple independent sketches and when doing recovery, we combine the outcomes recovered from each independent sketch (as described below). As mentioned earlier if $\mathbf{ED}(x, y) = k$, there could be many different sets of k edit operations that transform x to y. To output edit operations consistently among independent runs of the sketching algorithm we will always opt for a *canonical* choice of the edit operations. The canonical choice prefers insertions into x over substitutions which in turn are preferred over deletions from x. This preference is applied on edit operations from left to right in x so the choice of edit operations corresponds to the left-most shortest path in the usual edit distance graph of x and y. Importantly, the canonical path is consistent under taking substrings of x and y. See Section 3.2 for details on the choice of the path.

Bad splits. For two strings x and y, as we split the fragments of each through successive levels we will inevitably split some fragments near an edit operation; this is called a *bad split*. A bad split might cause x to be divided at a location but not y. This causes two problems: (1) sub-fragments of the badly split fragment may not align, and (2) the bad split may cause fragments to the right of the badly split fragment to be misaligned. Both problems need to be dealt with.

Regularization of trees. To handle the second issue we regularize the decomposition tree. The decomposition tree is of depth at most $O(\log n)$ and it might have degree up to n. We make it regular as follows: for any node with fewer than n children we append

dummy children to the right of the real children; these are thought of as representing empty strings. We do this at all levels so that the tree has degree n in all the internal nodes and all the leaves are at depth exactly $d = O(\log k + \sqrt{\log n \log\log n})$. Each node of the tree is labeled by a fragment of x and its corresponding grammar. This ensures that the underlying tree for the decomposition is the same for every input string. Now, in constructing the level $i+1$ decomposition from the level i, the fragments that correspond to the children of node v in the two trees are paired with each other. A bad split arising from the decomposition of the two fragments corresponding to tree node v may cause misalignment among the descendants of v but will not affect the alignment of nodes to v's right.

Sparse representation. The regularized tree is of super-polynomial size $\leq n^{\log n}$, but only has $\widetilde{O}(n)$ nontrivial nodes that represent non-empty strings. As a result, the tree can be constructed in polynomial time, and represented concisely by the set of ordered pair of (node,fragment) for nonempty fragments. Furthermore the sketching algorithm works well with the sparse representation and its running time is at most polynomial in the number of non-trivial nodes.

Mismatch floods. The more significant issue caused by bad splits is the first one, that a bad split at a node may result in bad splits at many of its descendants. Indeed, if the partition of the fragments x_v and y_v of x and y at node v are not compatibly split, then the fragment pairs of its children will not be aligned and the edit distances between fragments corresponding to children of v may be arbitrarily large, This misalignment will propagate down the tree possibly resulting in huge edit distance between fragments for many nodes in the subtree rooted at v, and the grammar encodings of these fragments may have a very large Hamming distance.

A node in the tree that is in the subtree of a badly split node is referred to as *flooded* (with errors). At level i of the tree, we only need to recover the grammars corresponding to the unflooded nodes of the level (because the edit operations for eacy flooded node will be recovered by the sketches corresponding to an unflooded ancestor in the tree.) The usual mismatch recovery sketch does not allow for such selective recovery of unflooded portions, because the flooded portion may cause the total number of mismatches to far exceed the capacity of the mismatch recovery scheme. Thus we design a new variant of Hamming schemes, which we call *hierarchical mismatch recovery scheme*, that recovers differences in the unflooded parts.

Hierarchical mismatch recovery. The *hierarchical mismatch recovery scheme* is applied to a vector that is indexed by the leaves of a tree. The specification of the problem includes an assignment of a positive capacity κ_v to each node v in the tree, where all nodes at level i have the same capacity κ_i. For any two strings x and y, the capacity function induces a *load function* $\widehat{\kappa}$, where the value $\widehat{\kappa}_v$ is 1 on leaves where x and y differ and 0 on other leaves, and the load on a node v is the minimum of the sum of the loads of its children and its own capacity, A node v is *underloaded* if its load is less than κ_v/R for some parameter R, and a leaf is *accessible* if every node on its path is underloaded. The scheme is required to recover the mismatch information for all underloaded leaves. The sketch is implemented as follows: Let d be the depth of the tree. For each node at level $d-1$ we apply a (standard) mismatch recovery scheme to the vector of its children that handles (slightly more than) κ_{d-1} mismatches. Then working the way up the tree, for each node we apply a mismatch recovery to the vector consisting of the sketches computed at each of its children. This is passed upwards and the final sketch at the root is the output. The specific mismatch recovery scheme used is a a *superposition scheme* which is described in Section 5.2. The hierarchical sketches for two strings x and y will allow for recovery of mismatches occurring at all accessible leaves. The intuition behind this is as follows. For two strings x and y, label each node by the pair of intermediate sketches for x and y at that node. This pair of sketches encodes information about the mismatches of x and y at the leaves of its subtree. This information at node v is a compression to $\widetilde{O}(\kappa_i)$ bits of the information from its children. Inductively one can show that for an underloaded node the compressed string it computes has enough bits to preserve the information about accessible mismatches that is encoded in the sketches of its children. For overloaded nodes, the compression will destroy the information about its children, but this is not a problem because the scheme is not required to recover mismatch information for leaves below an overloaded node. An important thing to note is that the impact of overloaded nodes among v's children (which may have a large number of mismatches among its leaves) is controlled by the fact that the sketch size at those nodes is restricted by its capacity.

In our application to sketching the grammars we assign the capacity to each level of the tree so that the capacity is proportional to the grammar size k_i we expect at that level in the decomposition tree. Nodes that correspond to misaligned fragments (due to a bad split at an ancestor) and have a huge edit distance will correspond to overloaded nodes, and as discussed above, the hierarchical scheme contains the damage caused by the error flood to the subtree below the occurrence of the bad split. The idea behind the choice of the parameters is that the probability of a bad split is proportional to the number of edit operations in the fragment; it is roughly $\widetilde{O}(\#\text{ of edit op's}/k_i)$. Hence, in expectation each edit operation is responsible for $\widetilde{O}(1)$ mismatches resulting from bad splits that contribute to possible flooding of a node. We can adjust the parameters so that the flooding of a node is in expectation only a tiny fraction of its capacity. We can then apply Markov's inequality to argue that with a good probability nodes along a chosen path are not overloaded, i.e., flooded.

Details for our hierarchical mismatch recovery scheme are given in Section 5.1.

Parameters. For our edit distance sketch we will set the parameters as follows: $\kappa_0 = \widetilde{O}(s_{\mathrm{OR}}^2 k)$, $\kappa_i = \kappa_0/2^i$, $k_0 = \widetilde{O}(s_{\mathrm{OR}} k)$, $k_i = k_0/2^i$ and $t_0 = \widetilde{O}(s_{\mathrm{OR}} k)$, $t_i = t_0/2^i$. Our sketch will be obtained by applying the hierarchical mismatch recovery scheme to each level of the decomposition tree with those parameters.

Infrequent bad splits. The decomposition procedure is designed so that for a node v at level i with x and y fragments x_v and y_v, if $\mathbf{ED}(x_v, y_v) \leq k_i/C$ where $C = \mathrm{polylog}(k_i)$ the decompositions of x_v and y_v will be compatible with probability $1 - 1/\mathrm{polylog}(k_i)$. While this is near 1, it is likely a non-trivial fraction of nodes will fail to be compatibly split even though the edit distance between the strings is small compared to k_i. As a result, for any given edit operation, there may be a small but non-trivial chance that the

recovery algorithm fails to recover it. To ensure that we get all of the edit operations, we will need to run the scheme $O(\log n)$ times and include those edit operations produced by more than half the runs to guarantee that every edit operation gets recovered with good probability.

Location, location, location! The recovery procedure identifies pairs of fragments that differ, and can therefore reconstruct the canonical edits between those fragments. But as described so far, there is nothing that allows the reconstruction to pinpoint where these fragments appear in the full string. Without this information, we can not combine information obtained from the independent sketches just described. Our sketch will need an additional component to properly position each recovered fragment.

Location tree. We will use technique of Belazzougui and Zhang [3] (suggested to us by Tomasz Kociumaka). We turn the decomposition tree into a binary tree by expanding each node with n children into a binary tree of depth $\log(n)$ with n leaves. For each node in the binary tree, we record the length of the substring represented by its left child. We watermark this size by the usual Karp-Rabin fingerprint of the node substring, and we sketch the sizes using the hierarchical mismatch sketch as in the grammar tree, i.e., level by level. Fragments that contain edit operations will differ in the Karp-Rabin fingerprint so they will reveal the size of their left child. For a given node that contains an edit operation all of its ancestors on the path to the root will also be watermarked. Hence we will be able to recover the information about the size of all the left children along the path. That suffices to calculate the position of each differing fragment. We will use the same setting of capacities for the hierarchical mismatch scheme that we use for grammars. That is clearly sufficient as grammars are larger objects than a single integer.

Putting things together. The actual sketch consists of multiple independent sketches. Each of these sketches is the output of the hierarchical mismatch recovery scheme applied to each of the levels of the grammar decomposition tree, and applied to each of the levels of the binary location tree.

Recovery. We briefly explain the recovery of edit operations from sketches for two strings. The reconstruction starts by running the recovery procedure for all the hierarchical mismatch recovery sketches. This recovers various pairs of grammars with information about their location within the original input strings x and y. From those grammars we pick only those which do not have any ancestor grammar node recovered as well. (Descendant grammars are superseded by the ancestor grammars.) For every pair of grammars we reconstruct the fragments they represent and compute the associated edit operations. (The edit operations could be actually computed without decompressing the grammars [12].) For each pair of recovered fragments, we use the location tree sketches to recover the exact location of that fragment.

So for each pair of recovered fragments we calculate the canonical sequence of edit operations and given the exact location of the fragment, we can determine the exact location assign within x and y where each edit occurred. We repeat this for each independent copy of the sketch. Our final output is the set of edit operations that appear in the majority of the copies.

Due to space restrictions this extended abstract will only provide details for the main technical tool: the hierarchical mismatch recovery scheme. The full description of our edit distance sketch and its proof of correctness are in the full version of the paper that appears on arXiv.

3 PRELIMINARIES

3.1 Strings, Sequences, Trees, and String Decomposition

For an alphabet Γ, Γ^* denotes the set of (finite) strings of symbols from Γ, Γ^n is the strings of length exactly n and $\Gamma^{\leq n}$ is the set of strings of length at most n. We write ε for the empty string of length 0. The length of string x is denoted $|x|$. For an index $i \in \{1, \ldots, |x|\}$, x_i is the i-th symbol of x.

We will consider the input alphabet of our strings to be $\Sigma = \{0, 1, \ldots\}$ which is the set of natural numbers. For strings of length at most n we assume they are over sub-alphabet $\Sigma_n = \{0, 1, \ldots, n^3 - 1\}$ so typically we will assume our input comes from the set $\Sigma_n^{\leq n}$. This assumption is justified as for computing edit or Hamming distance of two strings of length at most n we can hash any larger alphabet randomly to the range Σ_n without affecting the distance of the two strings with high probability.

If Γ is linearly ordered then Γ^* is also linearly ordered under lexicographic order, denoted by $<_{\text{lex}}$, given by $x <_{\text{lex}} y$ if x is prefix of y or if $x_j < y_j$ where j is the least index i for which $x_i \neq y_i$. We usually write $x < y$ instead of $x <_{\text{lex}} y$.

We write $x \circ y$ for the concatenated string x followed by y and for a list $z_1, \ldots, z_k$ of strings we write $\odot_{i=1}^{k} z_i$ for $z_1 \circ \cdots \circ z_k$.

Substrings and fragments. For a string x and an interval $I \subseteq \{1, \ldots, |x|\}$, a string z is a *substring of x located at I* if $|z| = |I|$ and for all $i \in I$, $z_{i-\min(I)+1} = x_i$. We denote this substring by x_I. When using intervals to index substrings, it is convenient to represent intervals in the form $(i, j] = \{i + 1, \ldots, j\}$ and $(i, i]$ denotes the empty set for any i. (So a substring is always a consecutive subsequence of a string.) We can also say that z is the *substring of x starting at position* $\min(I)$. Furthermore, z is a *substring of x* if z is a substring of x starting at some position. However, the statement that z is a substring of x says nothing about where z appears in x, and there may be multiple (possibly overlapping) occurrences of z in x. For us it will be important where a substring appears. For a string x and an interval $I \subseteq \{1, \ldots, |x|\}$, the *fragment located at I* is the pair x_I together with I.

Sequences and Hamming Distance. We will consider finite sequences of elements from some domain. It will be convenient to allow sequences to have index sets other than the usual integers $\{1, \ldots, n\}$. If D is any set, a *D-sequence* is an indexed collection $a = (a_i : i \in D)$. A *D-sequence over the set A* is a D-sequence with entries in A. A^D denotes the set of D-sequences over A. The *Hamming Distance* $\mathbf{Ham}(x, y)$ between two D-sequences x and y is the number of indices $i \in D$ for which $x_i \neq y_i$. We let $I_{\neq}(x, y) = \{i \in D; x_i \neq y_i\}$.

Trees. Our algorithm will organize the processed data in a tree structure. To simplify our presentation we will give the tree very regular structure. For finite sets $L_1, \ldots, L_d$, $T(L_1 \times \cdots \times L_d)$ denotes the rooted tree of depth d where for each $j \in \{1, \ldots, d\}$ every internal node v at depth j has $|L_j|$ children, and the edges from v

to its children are labeled by distinct elements of L_j. Each node v at depth j is identified with the length j sequence of edge labels on the path from the root to v; under this correspondence the set of nodes at level j is $L_1 \times \cdots \times L_j$. The root is therefore the empty sequence ε. For an internal node v at depth $j-1$, its children are nodes of the form $v \circ a$ where $a \in L_j$. Also the path from ε to v at depth j is equal to the sequence of nodes $v_{\leq 0}, v_{\leq 1}, \ldots, v_{\leq j}$ where $v_{\leq i}$ is the prefix of v of length i.

Usually in this paper, the sets $L_1, \ldots, L_d$ are all equal to the same set L and in this case $T(L_1 \times \cdots \times L_d)$ is denoted $T(L^d)$. Usually, L is a linearly ordered set and it is useful to visualize the planar drawing of T in which the left-to-right order of the children of an internal node corresponds to the total ordering on the edge labels.

String decompositions, and tree decompositions. A *string decomposition* of a string x is a sequence $z_1, \ldots, z_r$ of strings such that $x = \odot_{i=1}^{r} z_i$. More generally if L is a linearly ordered set then an L-sequence $(z_i : i \in L)$ where each z_i is a string is a decomposition of x if $x = \odot_{i \in L} z_i$ where the concatenation is done in the order determined by L. Given a string decomposition z, each substring z_i is naturally associated to a location interval $\mathbf{Loc}(z)_i = (s_i, t_i]$ where $s_i = \sum_{j \in L, j<i} |z_j|$ and $t_i = s_i + |z_i|$.

Consider a string decomposition of string x whose substrings are indexed by the set L^d (in lexicographic order). Let us view L^d as the set of leaves of the tree $T(L^d)$ as defined above. We can label the leaves by the corresponding substrings of the decomposition, and then extend the labelling to the set $L^{\leq d}$ of all tree nodes. We identify the labeling of leaves with z, and extend it so for $v \in L^{\leq d}$, z_v is the concatenation, in lexicographic order, of the strings of all the leaves below it. Note also that z_v is the concatenation of the strings labeling the children of v. We refer to the labeling z as a *string decomposition tree* for x induced by the string decomposition z. Given a string decomposition tree, we extend the definition of location interval as $\mathbf{Loc}(z)_v = (s_v, t_v]$ where $s_v = \sum_{u \in L^d, u<v} |z_u|$ and $t_v = s_v + |z_v|$. Hence, $x_{\mathbf{Loc}(z)_v} = z_v$.

3.2 Edit Distance and its Representation in Grid Graphs

For $x \in \Sigma^*$, we consider three *edit operations* on x:

- **ins**(i, a) where $i \in \{1, \ldots, |x|+1\}$ and $a \in \Sigma$, which means insert a immediately following the prefix of length $i-1$. In the resulting sequence the i-th entry is a.
- **del**(i) where $i \in \{1, \ldots, |x|\}$, deletes the i-th entry of x.
- **sub**(i, b): replace x_i by b.

For strings x, y, the *edit distance of x and y*, $\mathbf{ED}(x, y)$, is the minimum length of a sequence of operations that transforms x to y. It is well-known and easy to show that $\mathbf{ED}(x, y) = \mathbf{ED}(y, x)$.

Representing edit distance by paths in weighted grids. We define **Grid** to be the directed graph whose vertex set $V(\mathbf{Grid})$ is the set $\mathbb{N} \times \mathbb{N}$ (*points*) and whose edge set $\mathbf{E}(\mathbf{Grid})$ consists of three types of directed edges: *horizontal edges* of the form $\langle i, j \rangle \to \langle i+1, j \rangle$, *vertical edges* of the form $\langle i, j \rangle \to \langle i, j+1 \rangle$ and *diagonal edges* of the form $\langle i, j \rangle \to \langle i+1, j+1 \rangle$ for any $i, j \geq \mathbb{N}$. For non-empty intervals $I, J \subseteq \mathbb{N}$ not-containing zero, we define the $\mathbf{Grid}_{I \times J}$ to be the subgraph of **Grid** induced on the set $(I \cup \{\min(I)-1\}) \times (J \cup \{\min(J)-1\})$, and for $P \subseteq E(\mathbf{Grid})$, the restriction of P to $I \times J$ is $P_{I \times J} = P \cap E(\mathbf{Grid}_{I \times J})$. We call $I \times J$ a *box*. A directed path from $\langle \min(I)-1, \min(J)-1 \rangle$ to $\langle \max(I), \max(J) \rangle$ is called a *spanning path* of $\mathbf{Grid}_{I \times J}$.

As is well known, the edit distance problem for a pair of strings x, y can be represented as a shortest path problem on a grid with weighted edges (see e.g. [18]). The *grid of x, y*, $\mathbf{Grid}(x, y)$, is the subgraph $\mathbf{Grid}_{(0,|x|] \times (0,|y|]}$ with edge set $\mathbf{E}(x, y) \subseteq \mathbf{E}(\mathbf{Grid})$. For an edge $e = \langle i, j \rangle \to \langle i', j' \rangle$ in $\mathbf{Grid}(x, y)$, let $x_e = x_{i'}$ if $i' = i+1$ and $x_e = \varepsilon$ otherwise. Similarly, let $y_e = y_{j'}$ if $j' = j+1$ and $y_e = \varepsilon$ otherwise. We assign a cost to edge e to be 0 if $y_e = x_e$ and it is 1 otherwise. In particular, every horizontal edge and every vertical edge costs 1, and diagonal edges cost 0 or 1 depending on whether the corresponding symbols of x and y differ. An edge of non-zero cost is *costly*. If P is a set of edges, the *costly part of P*, $\mathbf{costly}(P)$, is the set of costly edges. Define the cost of P to be $\mathrm{cost}(P) = |\mathbf{costly}(P)|$.

We define an *annotated edge* to be a triple (e, a, b) where $a, b \in \Sigma \cup \{\varepsilon\}$. The *$(x, y)$-annotation of e* is the annotated edge (e, x_e, y_e), which is denoted $e^+(x, y)$. An annotated edge (e, a, b) is said to be *(x, y)-consistent* or simply *consistent* if $a = x_e$ and $\beta = y_e$. We emphasize that each edge e has a unique consistent annotation (with respect to given x and y).

For a set of edges P we write $P^+(x, y)$ for the set of annotated edges $\{e^+(x, y) : e \in P\}$. For a path P, $\mathbf{costly}(P^+(x, y))$ is the set of costly edges of P with their x, y-annotations. When the pair x, y of strings is fixed by the context (which is almost always the case) we write e^+ for $e^+(x, y)$ and for a set P of edges, we write P^+ for $P^+(x, y)$. In particular, $\mathbf{E}^+$ for the set $\{e^+ : e \in \mathbf{E}(x, y)\}$.

It is well known and easy to see that there is a correspondence between spanning paths of $\mathbf{Grid}(x, y)$ and sequences of edit operations that transform x to y where a sequence of k edit operations corresponds to a spanning path of cost k. Thus, we will refer to a spanning path of $\mathbf{Grid}(x, y)$ as an *alignment of x and y*. We have:

Proposition 3.1. *$\boldsymbol{ED}(x, y)$ is equal to the minimum cost of an alignment of x and y.*

A pair x, y of strings together with a box $I \times J$ with $I \subseteq (0, |x|]$ and $J \subseteq (0, |y|]$ specifies the edit distance sub-problem $\mathbf{ED}(x_I, y_J)$. $\mathbf{Grid}_{I \times J}(x, y)$ denotes the (edge-weighted) sub-graph of $\mathbf{Grid}(x, y)$ induced on $(I \cup \{\min(I)-1\}) \times (J \cup \{\min(J)-1\})$.

There may be many optimal alignments. We will need a *canonical alignment* for each x and y that is unique. For any graph $\mathbf{Grid}_{I \times J}(x, y)$, define the *canonical alignment of $\boldsymbol{Grid}_{I \times J}(x, y)$* as follows: Associate each path in **Grid** to the sequence from {**vertical**, **diagonal**, **horizontal**} which records the edge types along the path. The *canonical alignment of $\boldsymbol{Grid}_{I \times J}(x, y)$* is the optimal spanning path of $\mathbf{Grid}_{I \times J}(x, y)$ that is lexicographically maximum with respect to the order **vertical** > **diagonal** > **horizontal**. The *canonical alignment $\boldsymbol{canon}(x, y)$ of x and y* is the canonical alignment of $\mathbf{Grid}(x, y)$.

The proof of the following is left to the reader.

Proposition 3.2. *For strings x, y and box $I \times J \subseteq (0, |x|] \times (0, |y|]$, the (edge-weighted) graph $\boldsymbol{Grid}_{I \times J}(x, y)$ is isomorphic (in the graph theoretic sense) to $\boldsymbol{Grid}(x_I, y_J)$ and so $\boldsymbol{ED}(x_I, y_J)$ is equal to the length of the shortest spanning path of $\boldsymbol{Grid}_{I \times J}(x, y)$. Also, the canonical alignment of x_I and y_J is isomorphic to the canonical alignment of $\boldsymbol{Grid}_{I \times J}(x, y)$ (when viewed as paths of their respective graphs).*

Let P be an alignment of x and y. A box $I \times J$ is *compatible with* P provided that P passes through $\langle \min(I)-1, \min(J)-1\rangle$ and $\langle \max(I), \max(J)\rangle$, and for such a box, the *restriction* $P_{I\times J}$ *of* P *to* $I \times J$ is the portion of P joining $\langle \min(I)-1, \min(J)-1\rangle$ and $\langle \max(I), \max(J)\rangle$. This restriction is an alignment for $\mathbf{Grid}_{I\times J}(x, y)$. The following proposition is straightforward.

Proposition 3.3. *If P is an optimal alignment of x and y and $I \times J$ is compatible with P then $P_{I\times J}$ is an optimal alignment for the sub-problem $\boldsymbol{Grid}_{I\times J}(x, y)$.*

For strings x and y, we say a box $I \times J$ is (x, y)-compatible if $I \times J$ is compatible with the canonical alignment $\mathbf{canon}(x, y)$. We have:

Proposition 3.4. *Let x, y be strings and let $I \times J$ be a box that is (x, y)-compatible. The restriction $\boldsymbol{canon}(x, y)_{I\times J}$ is equal to the canonical alignment of $\boldsymbol{Grid}_{I\times J}(x, y)$.*

4 THREE AUXILIARY PROCEDURES

Our sketching scheme for edit distance uses several auxiliary sketching schemes. Three of these are from previous work and one is new to this paper. In this section, we describe the key properties of the previous three schemes.

4.1 Fingerprinting

The *fingerprinting problem* for a Σ^* is to define a family of sketching functions that distinguishes between distinct elements of Σ^*. More formally, given a parameter n we want a family $\textsc{fingerprint}_\rho(\cdot; n) : \Sigma^{\le n} \to \mathbb{Z}^+$ such that for any strings $x, y \in \Sigma^{\le n}$ with $x \neq y$

$$\mathbf{Pr}_\rho[\textsc{fingerprint}_\rho(x; n) = \textsc{fingerprint}_\rho(y; n)] < 1/n^4.$$

The following classic result of Karp-Rabin [15] provides an efficient fingerprinting scheme. We refer to it as the *Karp-Rabin fingerprint*.

Theorem 4.1. *There is an efficiently computable randomized function $\textsc{fingerprint}_\rho(\cdot; n) : \Sigma^{\le n} \to \{1, \ldots, n^5\}$ such that for any $x, y \in \Sigma^{\le n}$ with $x \neq y$, $\boldsymbol{Pr}_\rho[\textsc{fingerprint}_\rho(x; n) = \textsc{fingerprint}_\rho(y; n)] < 1/n^4$. The number of bits needed to describe ρ is $O(\log n)$. The time to compute $\textsc{fingerprint}_\rho(x; n)$ is $O(|x| \cdot \log^{O(1)} n)$.*

4.2 Threshold Edit Distance Fingerprinting

In the *threshold edit distance fingerprinting* we are given a parameter n and a threshold parameter k. We want a family of sketching functions $\textsc{ted-fingerprint}_\rho(\cdot; k, n) : \Sigma^* \to \mathbb{Z}^+$ such that for all $x, y \in \Sigma^{\le n}$ such that $\mathbf{ED}(x, y) \ge k$, $\mathbf{Pr}[\textsc{ted-fingerprint}_\rho(x; k, n) = \textsc{ted-fingerprint}_\rho(y; k, n)] \le 1/n^4$, i.e. the sketch is very likely to distinguish strings x, y that are far. We also want that for some *inaccuracy gap* $s > 1$, for strings x, y such that $\mathbf{ED}(x, y) < \frac{k}{s}$, $\mathbf{Pr}[\textsc{ted-fingerprint}(x) \neq \textsc{ted-fingerprint}(y)]$ is small. Precisely we want that for all x, y:

$$\mathbf{Pr}[\textsc{ted-fingerprint}_\rho(x) \neq \textsc{ted-fingerprint}_\rho(y)] \le \frac{\mathbf{ED}(x, y)}{k} \cdot s.$$

Note that the requirement becomes easier as s gets larger. The problem of constructing a threshold edit distance fingerprinting scheme with inaccuracy gap $s > 1$ is closely related to the problem of finding an approximate embedding function f that maps strings to vectors in $\mathbb{R}^d$ (for some d) so that $\mathbf{ED}(x, y)$ is approximated by $|f(x) - f(y)|_1 = \sum_{i=1}^d |f(x) - f(y)|$ within a factor s. This latter problem was investigated by Ostrovsky and Rabani [20] and their results yield the following consequence:

Theorem 4.2. *There is an algorithm OR-$\textsc{fingerprint}(x; k, n)$ for the threshold edit distance fingerprinting problem that for arbitrary $k, n \in \mathbb{N}$ has inaccuracy gap $s_{\mathrm{OR}} = 2^{O(\sqrt{\log(n)\log\log(n)})}$ and for any string $x \in \Sigma^{\le n}$ gives a fingerprint of value at most $O(n^4)$. The fingerprinting algorithm uses a randomness parameter of length $O(\log^3 n)$ and runs in time polynomial in n and k.*

We refer to OR-$\textsc{fingerprint}(x; k, n)$ as Ostrovsky-Rabani fingerprint.

We make a few remarks on this theorem.[2] [20] does not provide explicit bound on the amount of randomness needed. In particular, Lemma 9 samples random subsets and uses tail-bound inequalities to bound the probability of bad events. One can use $O(\log n)$-wise independent samples to reduce the necessary randomness [11].

4.3 The Procedure BASIC-DECOMP

The sketch for edit distance from [4] gave sketches of size $\widetilde{O}(k^2)$. A key procedure in their algorithm is an essential part of our sketch-and-recover scheme. We restate the properties of algorithm BASIC-DECOMP in a form suitable for us. The algorithm takes as input a string x, parameter n and an integer *sparsity parameter* k, where $|x| \le n$. We denote this by $\textsc{basic-decomp}(x; n, k)$.

$\textsc{basic-decomp}(x; n, k)$ is either UNDEFINED or outputs:

- A string decomposition of x, $z = (z_i : i \in W)$, where W is the set $\{0, 1\}^{\lceil \log n \rceil}$ with the lexicographic order. (Some strings may be empty.)
- A collection $\mathbf{grams} = (\mathbf{grams}_i : i \in W)$ of k-sparse bit-vectors (a bit-vector is t sparse if it has at most t 1's) of length $N = n^{60}$ such that $\textsc{basic-decode}(\mathbf{grams}_i) = z_i$ if $z_i \neq \varepsilon$ (so $\mathbf{grams}_i$ is a k-sparse encoding of z_i) and if $z_i = \varepsilon$ then $\mathbf{grams}_i$ is the all 0 vector 0^N. Furthermore, $\mathbf{grams}_i$ has the following *minimality property*: for any bit-vector b that is bit-wise less than $\mathbf{grams}_i$, $\textsc{basic-decode}(b)$ is UNDEFINED. (For technical ease we require $\textsc{basic-decode}(0^N)$ to also be UNDEFINED although 0^N represents the empty string.) Each $\mathbf{grams}_i$ is represented as a list of positions that are set to 1.

Theorem 4.3. *The algorithm BASIC-DECOMP has the following properties for all n, k:*

(1) For any input x, the probability that $\textsc{basic-decomp}(x; n, k)$ is UNDEFINED is at most $\frac{1}{n^4}$. The running time of $\textsc{basic-decomp}(x; n, k)$ as well as the total number of ones in $\boldsymbol{grams}_i$'s is bounded by $O(|x| \cdot \log^{O(1)} n)$.

(2) For any pair of inputs x, y for which the $\textsc{basic-decomp}(x; k)$ and $\textsc{basic-decomp}(y; k)$ is not UNDEFINED, suppose $z = (z_i : i \in W)$ and $\boldsymbol{grams} = (\boldsymbol{grams}_i : i \in W)$ is the output on x

[2] We had difficulties reading the paper [20]. Indeed, there is a minor correctable issue in the way Lemmas 8 and 9 are presented in the paper. The lemmas claim output from ℓ_1 which allows for vectors with arbitrary real numbers. Indeed, because of the scaling in their proofs they do output vectors with real numbers. However, both lemmas need to be applied iteratively and they assume input to be a 0-1-vector. This disparity can be corrected using standard means but it requires additional effort.

and $z' = (z'_i : i \in W)$ *and* $\boldsymbol{grams}' = (\boldsymbol{grams}'_i : i \in W)$ *is the output on* y.

(a) *With probability at least* $1 - s_{\text{split}} \cdot \frac{\boldsymbol{ED}(x,y)}{k}$ *where* $s_{\text{split}} = O(\log^4 n)$, *for all* $i \in W$, *the box* $\boldsymbol{Loc}(z)_i \times \boldsymbol{Loc}(z')_i$ *is* (x,y)*-compatible whenever* z_i *is non-empty, and* z_i *is non-empty iff* z'_i *is non-empty.*

(b) *For all* $i \in W$, $\boldsymbol{Ham}(\boldsymbol{grams}_i, \boldsymbol{grams}'_i) \le s_{\text{E}\to\text{H}} \cdot \boldsymbol{ED}(z_i, z'_i)$ *where* $s_{\text{E}\to\text{H}} = O(\log^2 n)$.

The point of Item 2a is that if for each $i \in W$, the box $\mathbf{Loc}(z)_i \times \mathbf{Loc}(z')_i$ is (x,y)-compatible then $\mathbf{ED}(x,y) = \sum_{i\in W} \mathbf{ED}(z_i, z'_i)$.

Note, for Item 1, [4] claims success probability of decomposition to be only $O(1/n)$ so we apply their construction with their parameter n to be set to our n^4. For Item 2a, they do not specify the success probability explicitly the way we do but our formula is immediate from their analysis of probability of what they call *undesirable split*. The upper bound on the difference between two grammars in terms of edit distance of x and y is also implicit in the same analysis.

Given a grammar G by its sparse representation as a list of t positions of 1's, BASIC-DECODE(G) runs in time $O(t+|x|)$ if G represents a string x and in time $O(t)$ otherwise. A grammar representing x contains at most $s_{\text{E}\to\text{H}} \cdot |x|$ ones.

5 SKETCH-AND-RECOVER SCHEMES

A *sketch-and-recover* scheme consists of two algorithms: a sketching algorithm that takes a string x over some alphabet and produces a shorter string $\text{SK}(x)$ (a *sketch*) perhaps over a different alphabet, and a *recovery algorithm* that takes two sketches $\text{SK}(x)$ and $\text{SK}(y)$ and recovers a sequence of edit operations that turn x into y and vice-versa.

The schemes are randomized and typically take some list ℓ of auxiliary parameters (such as the maximum input length handled by the sketch). Thus the sketch $\text{SK}_\rho(x;\ell)$ is a randomized function. The recovery algorithm requires that $\text{SK}_\rho(x;\ell)$ and $\text{SK}_\rho(y;\ell)$ were created using the same randomizing parameter ρ, which we think of as coming from public randomness accessible to all, and the same ℓ, and it needs to know ρ. We allow the sketch function on a given input to fail (output UNDEFINED) with a small probability (with respect to ρ), and similarly for the recovery algorithm. We usually suppress ρ and ℓ and write simply $\text{SK}(x)$.

Efficiency of sketch-and-recover schemes is measured by:

- The *sketch length* bit-length$(\text{SK}_\rho(x;k,n))$, which is the number of bits in the binary encoding of $\text{SK}_\rho(x;k,n)$.
- The *parameter length* bit-length(ρ) is the number of bits (the binary encoding of) the parameter ρ.
- The running time of the sketch and recovery functions.

Our primary focus here is on achieving small sketch length. We also want the running time of the sketch and recovery algorithms to be at most polynomial in the length of the strings being sketched, but in this paper we are not trying to optimize the polynomial.

Our goal in this paper is a sketch-and-recover scheme that allows for recovery of edit distance $\mathbf{ED}(x,y)$ given the sketches of x and y. For this problem the list ℓ of sketching parameters is n,k were n is an upper bound on the length of strings and k is an upper bound on $\mathbf{ED}(x,y)$ for which the recovery algorithm must find the distance.

For Hamming distance sketches, there are known sketches that recover not only $\mathbf{Ham}(x,y)$ but also the *mismatch information* for x, y, $\text{MIS}(x,y)$, which is the set $\{(i, x_i, y_i) : i \in I_{\neq}(x,y)\}$. Note that given $\text{MIS}(x,y)$ and either string one can recover the other string.

The analog of mismatch information for Hamming distance is the set of costly annotated edges of the canonical alignment (see Section 3.2), which is enough to determine the canonical alignment

5.1 Hierarchical Mismatch Recovery

We will need sketch-and-recover schemes for Hamming distance over the set of D-sequences over some alphabet Γ that recover the mismatch information $\text{MIS}(u,w)$ defined in Section 5. We refer to this as *mismatch recovery*. We will need a more general formulation of mismatch recovery in which we only require recovery of mismatch information for certain indices. Those indices will be determined for each pair of strings u and w separately.

This more general formulation is called *targeted mismatch recovery*. Before giving a formal definition, we motivate it with an example. Consider sequences of length $n = m^2$ for some integer m, where we think of a sequence u as consisting of m fragments $u^1, \ldots, u^m$ each of length m, where fragment i is the substring $u_{(m(i-1),mi]}$. For strings u, w we say that fragment i is overloaded if there is more than one mismatch in the fragment, and underloaded if there is at most one mismatch in the fragment. Let $F(u,w)$ be the set of mismatch indices belonging to underloaded fragments. The set $F(u,w)$ defines a targeted mismatch recovery problem where our goal is to provide sketching and recovery algorithms so that for any pair $u, w \in \Gamma^D$, given the sketches of u and w, the recovery algorithm finds all mismatch indices belonging to $F(u,w)$. This particular example can be solved using a sketch of size $\widetilde{O}(m)$ as it will become clear later. One needs to recover at most m mismatches in underloaded fragments while not being distracted by the overloaded fragments.

In general, a *targeted mismatch recovery* problem is specified by a target function F which for each pair $u, w \in \Gamma^D$, satisfies $F(u,w) \subseteq I_{\neq}(u,w)$. The targeted mismatch recovery problem for a target function F is denoted $\mathbf{TMR}(F)$. The output of the RECOVER function applied to two sketches is required to be a set of triples (i,a,b) where $i \in D$ and $a, b \in \Gamma$. Such a triple is called a *mismatch triple*. The success conditions for $\mathbf{TMR}(F)$ for the pair u, w are:

Soundness. $\text{RECOVER}(\text{SK}(u), \text{SK}(w)) \subseteq \text{MIS}(u,w)$, i.e. every mismatch claim is correct.

Completeness. $F(u,w) \subseteq \text{RECOVER}(\text{SK}(u), \text{SK}(w))$ so the algorithm recovers all mismatch triples from $F(u,w)$.

Notice, we do not require that $\text{RECOVER}(\text{SK}(u), \text{SK}(w)) \subseteq F(u,w)$. Indeed, the recovery algorithm by itself might not know $F(u,w)$. So the Completeness depends on the target F but Soundness does not. A scheme for targeted mismatch recovery with target function F has failure probability at most δ provided that for any $u, w \in \Gamma^D$, with probability at least $1 - \delta$, both Completeness and Soundness hold. Here, the probability is taken over the randomness of the sketching and recovery algorithms. (Our recovery algorithms will actually be deterministic, so all the randomness is coming from the sketch.)

Hierarchical mismatch recovery (**HMR**) is a special case of targeted mismatch recovery where the target function depends on a

capacitated tree $(T(L^d), \kappa)$ which consists of a rooted tree $T(L^d)$ and an integer-valued capacity function κ on levels of the tree. For an integer j we write κ_j for the capacity of nodes at level j so we think of $\kappa = (\kappa_0, \ldots, \kappa_d)$. We require $1 = \kappa_d \le \kappa_{d-1} \le \cdots \le \kappa_0$.

We use $(T(L^d), \kappa)$ to formulate a targeted mismatch recovery problem for L^d-sequences over Γ. The target set for u and w depends on $(T(L^d), \kappa)$ and an *overload parameter* R, and is denoted $F_{T(L^d),\kappa,R}(u, w)$. We define it next. The *load function* $\widehat{\kappa}(u, w)$ associated to κ and a pair u, w is defined on the vertices of the tree $T(L^d)$ as follows:

- For a leaf $v \in L^d$, $\widehat{\kappa}_v = 1$ if $u_v \neq w_v$, and is 0 otherwise.
- For an internal node v at level $j < d$,

$$\widehat{\kappa}_v = \min(\kappa_j, \sum_{v' \in \mathbf{child}(v)} \widehat{\kappa}_{v'}).$$

Clearly $\widehat{\kappa}_v \le \kappa_j$ for every vertex v at level j. We say that an internal node v at level j is *R-overloaded*, for parameter $R > 1$, if $\widehat{\kappa}_v \ge \frac{1}{R}\kappa_j$ and is *R-underloaded* otherwise. (Later we will fix the parameter R to be $4d$, and refer to nodes simply as overloaded or underloaded.)

Intuitively, the leaves below a R-overloaded node are "crowded" with mismatches so we do not require them to be recovered. Notice that a R-underloaded v satisfies $\widehat{\kappa}_v = \sum_{v' \in \mathbf{child}(v)} \widehat{\kappa}_{v'}$. A leaf is said to be *R-accessible* if its path to the root consists entirely of R-underloaded nodes.

We define $F_{T(L^d),\kappa,R}(u, w)$ to be the set R-accessible leaves ℓ where $u_\ell \neq w_\ell$. We denote by $\mathbf{HMR}(T(L^d), \kappa, R)$ the targeted mismatch recovery problem with target function $F_{T(L^d),\kappa,R}$. In Section 5.4 we will prove:

Theorem 5.1. *Let $T(L^d)$ be a level-uniform tree as defined in Section 3.1 and let $\kappa = (\kappa_0, \ldots, \kappa_d)$ be a capacity function $1 = \kappa_d \le \kappa_{d-1} \le \cdots \le \kappa_0 \le |L^d|$. Let $|L^d|$ and all κ_j be powers of two. Let $\Gamma = \mathbb{F}_p$ where $p \ge 4|L^d|^2$. There is a sketch-and-recover scheme for hierarchical mismatch recovery for L^d-sequences over Γ defined by procedures* HMR-SKETCH *and* HMR-RECOVER *that given $\delta > 0$ satisfies:*

1. *The scheme solves $\mathbf{HMR}(T(L^d), \kappa, R)$ for any $R \ge 4d$ with failure probability at most δ.*
2. *The sketch bit-size is $O(\kappa_0 \cdot \log|\Gamma| \cdot (\log|L^d| + \log(1/\delta)))$ bits.*
3. *The sketching algorithm runs in time $O(|L^d| \cdot \log^{O(1)}|\Gamma| \cdot \log(1/\delta))$.*
4. *The recovery algorithm runs in time $O(\kappa_0 \cdot \log^{O(1)}|\Gamma| \cdot \log(1/\delta))$.*
5. *If the L^d-sequence u is given via the sparse representation $\{(j, u_j) : j \in \mathbf{supp}(u)\}$ where $\mathbf{supp}(u) = \{j \in L^d : u_j \neq 0\}$ then the time to construct the sketch is $O((\kappa_0 + |\mathbf{supp}(u)|) \cdot \log^{O(1)}|\Gamma| \cdot \log(1/\delta))$.*
6. *The number of mismatch pairs output by the algorithm is at most the capacity κ_0 of the root.*

The function HMR-SKETCH depends on $T(L^d)$, κ, R, and δ and so these are input parameters to HMR-SKETCH. In what follows we write HMR-SKETCH$(u; T(L^d), \kappa, R, \delta)$ for the sketch of u for the capacitated tree $(T(L^d), \kappa)$ and overload parameter R, and error parameter δ, and HMR-RECOVER(u, w) for the recovery function which takes as input the sketches u and w' output by HMR-SKETCH on two strings.

In our application we will also use a hierarchical mismatch recovery scheme for the special case of the tree $T(\{0,1\}^d)$ where $d = 0$. In that case we will need to sketch only a single value from Γ so we assume that its sketch consists of the value itself. The recovery procedure from two such sketches is straightforward.

We remark that the theorem easily generalizes to the case where the edges labeling edges at level i come from a set L_i with the sets $L_1, L_2, \ldots, L_d$ being different.

5.2 Superposition Sketch-and-Recover Schemes

This subsection and the next give a general approach to targeted mismatch recovery. We apply this approach in Section 5.4 to construct the sketch-and-recover scheme HMR-SKETCH and HMR-RECOVER for hierarchical mismatch recovery and prove Theorem 5.1.

We make the following assumptions:

- $D = \{0, \ldots, |D| - 1\}$.
- Γ is the field $\mathbb{F}_p$ for some prime larger than $|D|$ so $D \subseteq \Gamma$.

If these assumptions do not hold we can often reduce our situation to one where it does hold. We do not need D to be integers; it is enough that there is an easily computable 1-1 mapping m from D to the nonnegative integers. Letting $m = \max_{j \in D} m(j) + 1$, we can think of D as a subset of $\{0, \ldots, m-1\}$, and enlarge the domain to $\{0, \ldots, m-1\}$ defining any D-sequence to be 0 on those indices outside of D. Similarly we can replace the range Γ by $\mathbb{F}_p$ for some p whose size is at least $\max\{|\Gamma|, |D|\}$, where we interpret Γ as a subset of $\mathbb{F}_p$ via some easily computable 1-1 map.

Let u, w be D-sequences over Γ. Here we use a basic technique from [21] (see also [8, 9] for related constructions), that allows for the recovery of $\mathrm{MIS}(u, w)$ at a specific index i. For a parameter $\alpha \in \Gamma$, the *trace* of $u \in \Gamma^D$ (with respect to α), denoted $\mathbf{tr}_\alpha(u)$ is the D-sequence over Γ^4 where for each $i \in D$, $\mathbf{tr}_\alpha(u)_i$ has entries:

$$\begin{aligned}
\mathbf{tr}_\alpha(u)_{i,\text{value}} &= u_i, \\
\mathbf{tr}_\alpha(u)_{i,\text{product}} &= i \cdot u_i, \\
\mathbf{tr}_\alpha(u)_{i,\text{square}} &= u_i^2, \\
\mathbf{tr}_\alpha(u)_{i,\text{hash}} &= \alpha^i u_i.
\end{aligned}$$

All the calculations are done over $\mathbb{F}_p$. We refer to a vector in Γ^4 with indices from {value, product, square, hash} as a *trace vector* and we refer to α as the *trace parameter*

For $u, w \in \Gamma^D$, the *trace difference of u, w* is $\Delta_\alpha(u, w) = \mathbf{tr}_\alpha(u) - \mathbf{tr}_\alpha(w)$. Here, for each i, $\Delta_\alpha(u, w)_i = \mathbf{tr}_\alpha(u)_i - \mathbf{tr}_\alpha(w)_i$ is a trace vector obtained by coordinate-wise subtraction.

Define the function **restore** which maps trace vectors t to Γ^3 as follows:

$$\begin{aligned}
\mathbf{restore}(t)_{\text{index}} &= \frac{t_{\text{product}}}{t_{\text{value}}} \\
\mathbf{restore}(t)_{\text{x-val}} &= \frac{t_{\text{square}} + t_{\text{value}}^2}{2t_{\text{value}}} \\
\mathbf{restore}(t)_{\text{y-val}} &= \frac{t_{\text{square}} - t_{\text{value}}^2}{2t_{\text{value}}}.
\end{aligned}$$

It is easy to check:

Proposition 5.2. *The mismatch information for (u, w) at i is completely determined by $\Delta_\alpha(u,w)_i$ as follows: i is a mismatch index of u, w if and only if $\Delta_\alpha(u,w)_i \neq 0$ and for such an i,*

$$\begin{aligned} \textbf{\textit{restore}}(\Delta_\alpha(u,w))_{i,index} &= i, \\ \textbf{\textit{restore}}(\Delta_\alpha(u,w))_{i,x\text{-}val} &= u_i, \\ \textbf{\textit{restore}}(\Delta_\alpha(u,w))_{i,y\text{-}val} &= w_i, \end{aligned}$$

and therefore $\textbf{\textit{restore}}(\Delta_\alpha(u,w))_i = (i, u_i, w_i) = \mathrm{MIS}(u,w)_i$.

We remark that the division by 2 in the definition of **restore** is the reason why we need that Γ does not have characteristic 2. Also, note that $\mathbf{tr}_\alpha(u)_{i,\mathrm{hash}}$ is not used in **restore**, but is used later to check soundness. In standard binary representation of integers, all arithmetic operations over $\mathbb{F}_p$ that are necessary to compute trace or its restoration at a single coordinate can be computed in time $O(\log^{O(1)} p)$.

We are now ready to define the class of *superposition sketches* for functions from D to Γ.

Definition 5.3 (Superposition sketch). Let S be a set, $h : D \to S$ and $\alpha \in \Gamma$. The *superposition sketch induced by* (α, h) is the function $\mathbf{tr}_{\alpha,h}$ that maps $u \in \Gamma^D$ to $\mathbf{tr}_{\alpha,h}(u) \in (\Gamma^4)^S$ where for $j \in S$:

$$\mathbf{tr}_{\alpha,h}(u)_j = \sum_{i \in h^{-1}(j)} \mathbf{tr}_\alpha(u)_i.$$

In words, the function h is used to partition D into $|S|$ classes $h^{-1}(j)$, and $\mathbf{tr}_{\alpha,h}(u)$ at $j \in S$ is the sum of the trace vectors of u corresponding to indices of D in the class $h^{-1}(j)$. The size (in bits) of the output is $O(|S| \log |\Gamma|)$.

Note that we can compute the superposition sketch of any D-sequence over Γ easily: Initialize $\mathbf{tr}_{\alpha,h}(u)$ to all zero and then for each $i \in D$ add the trace vector $\mathbf{tr}_\alpha(u)_i$ to $\mathbf{tr}_{\alpha,h}(u)_j$ where $j = h(i)$.

For $u, w \in \Gamma^D$ and $h : D \to \Gamma$, a mismatch index $i \in D$ is *recoverable for u, w, h* if $h^{-1}(h(i)) \cap I_{\neq}(u,w) = \{i\}$. We now define a procedure $\textsc{recover}_{\alpha,h}$ that recovers all recoverable indices.

$\textsc{recover}_{\alpha,h}(\mathbf{tr}_{\alpha,h}(u), \mathbf{tr}_{\alpha,h}(w), \alpha, h)$

Input: Traces $\mathbf{tr}_{\alpha,h}(u), \mathbf{tr}_{\alpha,h}(w)$ for two strings $u, w \in \Gamma^D$, trace parameters $\alpha \in \Gamma$, $h : D \to S$.

Output: The set $M_{\alpha,h}(u,w)$ of mismatch triples.

---

1 Let $\Delta_{\alpha,h}(u,w) = \mathbf{tr}_{\alpha,h}(u) - \mathbf{tr}_{\alpha,h}(w)$.

2 Let J be the set of $j \in S$ such that $\Delta_{\alpha,h}(u,w)_{j,\mathrm{value}} \neq 0$.

3 **Rebuilding step:** For each $j \in J$ let $z_j = \mathbf{restore}(\Delta_{\alpha,h}(u,w)_j)$.

4 **Filtering step:** Let $I = \{j \in J,\ z_{j,\mathrm{index}} < |D| \ \wedge\ \Delta_{\alpha,h}(u,w)_{j,\mathrm{hash}} = \alpha^{z_{j,\mathrm{index}}}(z_{j,x\text{-val}} - z_{j,y\text{-val}})\}$.

5 Return $M_{\alpha,h}(u,w) = \{z_j : j \in I\}$.

In the rebuilding step, the algorithm produces a list of mismatch triples by applying **restore** to every trace vector it can among the $\Delta_{\alpha,h}(u,w)_j$. In the filtering step, it eliminates some of these mismatch triples, and then it outputs the rest. The following lemma shows that (1) The rebuilding step produces all mismatch triples corresponding to recoverable indices (and possibly some others) and (2) The filtering step with high probability eliminates all mismatch triples corresponding to indices that are not recoverable, so Soundness holds with high probability.

Lemma 5.4. *Let $u, w \in \Gamma^D$ and $h : D \to S$ be fixed. Let I and J are as in the $\textsc{recover}_{\alpha,h}(u,w)$. For each $j \in S$:*

(1) If $|h^{-1}(j) \cap I_{\neq}(u,w)| = 0$ then $j \notin J$.
(2) If $|h^{-1}(j) \cap I_{\neq}(u,w)| = 1$ then $z_j \in M_{\alpha,h}(u,w)$ and $z_j = (i, u_i, w_i)$ where i is the unique mismatch index such that $h(i) = j$.
(3) The probability that $M_{\alpha,h}(u,w)$ outputs a triple that is not in $\mathrm{MIS}(u,w)$ (i.e. that Soundness fails) is at most $\frac{(|D|-1)\cdot|S|}{|\Gamma|}$ over a uniformly random choice of $\alpha \in \Gamma$.

The following result gives upper bounds on the running time of the sketch and recover algorithms, and on the space needed for the sketch

Proposition 5.5. *Let $\Gamma = \mathbb{F}_p$ and let $D = \{0, \ldots, |D|-1\}$ with $|D| \leq p$. Let S be a set, $h : D \to S$ and $\alpha \in \Gamma$. The superposition sketch $\textbf{\textit{tr}}_{\alpha,h}$ maps a D-sequence u over Γ to a sketch of bit-length $O(|S| \log |\Gamma|)$. The running time for the sketch algorithm is $O(|D| \cdot T)$ and the running time of the recover algorithm is $O(|S| \cdot T)$, where T is an upper bound on the time to perform a single arithmetic operation over Γ and evaluate h at a single point.*

Furthermore if a D-sequence u over Γ is given via a sparse representation, via $\{(j, u_j) : j \in \textbf{\textit{supp}}(u)\}$ where $\textbf{\textit{supp}}(u) = \{j \in D : u_j \neq 0\}$ then the running time of the sketch algorithm is $O((|S| + |\textbf{\textit{supp}}(u)|) \cdot T)$.

5.3 Randomized Superposition Sketches

In order to apply the superposition sketch we need to select a good h. However, one can hardly hope that if S is comparable in size to $\mathrm{MIS}(u,w)$ then one can find a single $h : D \to S$ for which all mismatch indices $I_{\neq}(u,w)$ will be recoverable. Hence, we will try superposition sketches for *multiple randomly chosen h*'s. Fix a (small) family $H \subseteq \{h : D \to S\}$ and a probability distribution μ on H (not necessarily uniform). For $\beta \leq 1$, we say that i is *β-recoverable for u, w, μ* provided that for $h \sim \mu$, the probability that i is recoverable for u, w, h is at least β. (Recall that if i is recoverable for u, w, h then for any choice of α, the output of recovery procedure from the sketches $\mathbf{tr}_{\alpha,h}(u)$ and $\mathbf{tr}_{\alpha,h}(w)$ includes the triple (i, u_i, w_i).)

We select hash functions $h_1, \ldots, h_\ell$ independently according to μ, for some *redundancy parameter* ℓ. We also select trace parameters $\alpha_1, \ldots, \alpha_\ell$ uniformly at random from Γ. The sketch of u consists of the sequences $h_1, \ldots, h_\ell$ and $\alpha_1, \ldots, \alpha_\ell$ together with $\mathbf{tr}_{\alpha_1,h_1}(u), \ldots, \mathbf{tr}_{\alpha_\ell,h_\ell}(u)$. For the recovery algorithm, given the sketches for u and w we compute each of the sets $M_{\alpha_i,h_i}(u,w)$ for $i \in [\ell]$ and define $M_{\alpha_1,\ldots,\alpha_\ell,h_1,\ldots,h_\ell}(u,w)$ to be the set of triples that appear in strictly more than half of the sets.

We refer to a scheme of the above type as a *randomized superposition scheme*. Such a scheme is determined by the distribution (H, μ) over hash functions and the redundancy ℓ.

The size of the sketch in bits (not including the description of the hash functions used) is $O(\ell \cdot |S| \cdot \log |\Gamma|)$. The description of the hash functions depends on the method used to represent members of H. For standard explicit choices of H (such as explicit families of

$O(1)$-wise independent functions), members of h are represented in $O(\log |H|)$ space.

Proposition 5.6. *Let D and S be sets and let Γ be a field of size at least $4(|D|-1)|\cdot|S|$. For each pair of strings $u, w \in \Gamma^D$, suppose $F(u,w)$ is a subset of D. Let (H,μ) be a distribution over hash functions from D to S. Suppose that for every u, w, every index belonging to $F(u,w)$ is 3/4-recoverable for u, w, μ. Then for any $\delta > 0$, the superposition sketch using (H,μ) with redundancy $\ell \geq 8(\ln|D| + \ln\frac{1}{\delta} + 2)$ satisfies the Completeness and Soundness conditions for F with failure probability at most δ.*

5.4 Proof of Theorem 5.1

In this section we show how to apply the randomized superposition schemes of Section 5.3 to construct a sketch-and-recover scheme for the hierarchical mismatch recovery and prove Theorem 5.1. The reader should review the set-up for hierarchical mismatch recovery in Section 5.1.

Recall that the scheme gets a capacitated tree $(T(L^d), \kappa)$ where $\kappa = (\kappa_0, \ldots, \kappa_d)$ and for each j, $\kappa_j \geq \kappa_{j+1}$. In preparation for describing the sketch-and-recover scheme, we associate to each node v at level j of $T(L^d)$, a set of *buckets* which are ordered pairs (v,i) where $1 \leq i \leq \kappa_j$. We refer to (v,i) as a v-bucket, and a bucket of the root is a *root-bucket*. Each leaf ℓ has only one bucket, $(\ell, 1)$.

The hash functions of our superposition scheme are **leaf→root** functions, which are functions that map L^d (the set of leaves) to the set of root-buckets. We will use Proposition 5.6 to prove that the scheme works by:

(1) Describing a distribution μ over **leaf→root** functions.
(2) Showing that for any pair of strings u, w, every leaf that belongs to $F_{T(L^d),\kappa,R}(u,w)$ is 3/4-recoverable for u, w, μ.

To describe the distribution μ on **leaf→root** functions, we consider a specific representation of a **leaf→root** functions, and for this we need the notions of a *trajectory* and *routing functions*.

For a leaf $\ell \in L^d$, the path from ℓ to the root ε in $T(L^d)$ are the nodes identified by $\ell_{\leq d}, \ell_{\leq d-1}, \ldots, \ell_0$. We define a *trajectory for ℓ* to be a sequence of buckets one for each node on the path from ℓ to ε, $(\ell_{\leq d}, i_d), (\ell_{\leq d-1}, i_{d-1}), \ldots, (\ell_0, i_0)$ where $i_j \in \{1, \ldots, \kappa_j\}$. A trajectory is uniquely determined by the leaf ℓ and the sequence of indices $(i_d, i_{d-1}, \ldots, i_0)$. Note that i_d must equal 1.

We want a way to specify a trajectory for every leaf. We do this using a collection $r = (r_j : j < d)$ of *routing functions*, one for each internal level of the tree. The routing function r_j is a function from $L \times \{1, \ldots, \kappa_{j+1}\}$ to $\{1, \ldots, \kappa_j\}$. For v at level j it maps buckets corresponding to children of v to buckets of v as follows: for $a \in L$ and $i \in \{1, \ldots, \kappa_{j+1}\}$, the bucket $(v \circ a, i)$ is mapped to $(v, r_j(a,i))$. Thus the collection of routing functions determines a trajectory for every leaf ℓ with sequence of indices $i_d(\ell) = 1$ and for $j < d$, $i_j(\ell) = r_j(\ell_{j+1}, i_{j+1}(\ell))$. This induces the **leaf→root** mapping that maps each $\ell \in L^d$ to the bucket $(\varepsilon, i_0(\ell))$.

We are now ready to specify the distribution μ. For each level $0 \leq j < d$, let $H_j = \{h : L \times \{1, \ldots, \kappa_{j+1}\} \to \{1, \ldots, \kappa_j\}\}$ be a pairwise independent family of routing functions for level j. Independently select $r_0, \ldots, r_{d-1}$ from $H_1, \ldots, H_{d-1}$. The distribution μ on **leaf→root** functions is the distribution induced by the selection of $r_0, \ldots, r_{d-1}$. The total number of bits to represent a **leaf→root** function in the family is $O(d(\log|L| + \log\kappa_0))$.

Lemma 5.7. *Let $(T(L^d), \kappa)$ be a capacitated tree. Let μ be the distribution on **leaf→root** functions induced by choosing routing functions $r_0, \ldots, r_{d-1}$ independently from pairwise independent distribution. For any two strings u, w in Γ^{L^d}, every $\ell \in F_{T(L^d),\kappa,4d}(u,w)$, i.e., every $4d$-accessible mismatch leaf, is 3/4-recoverable for u, w, μ.*

Proof. Let $\ell \in L^d$ be a $4d$-accessible leaf with respect to u, w where $u_\ell \neq w_\ell$. We must show that ℓ is 3/4-recoverable. Recall that ℓ is $4d$-accessible if each node along the path from ℓ to root is $4d$-underloaded, i.e., for each $j < d$, $\widehat{\kappa}_{\ell_{\leq j}} < \kappa_j/4d$. Let $(r_j : 0 \leq j < d)$ be the sequence of random routing functions selected as above and let f be the induced **leaf→root** function.

Let $(i_d, \ldots, i_0)$ denote the sequence of indices $(i_d(\ell), \ldots, i_0(\ell))$ for the trajectory of ℓ. This is a random variable depending on the choice of $r_0, \ldots, r_{d-1}$. By definition, ℓ is not recoverable if and only if there is a mismatch leaf $\ell' \neq \ell$ such that $f(\ell) = f(\ell')$. If $\ell' \neq \ell$ is a leaf such that $f(\ell') = f(\ell)$ then the trajectories of ℓ and ℓ' have non-empty intersection. We say that ℓ and ℓ' *merge at level j* if they are in different buckets at level $j+1$, but in the same bucket at level j. (Note that once the trajectories merge, they remain the same all the way to the root.)

For $j \in \{0, \ldots, d-1\}$, let MERGE_j be the event that there is a mismatch leaf $\ell' \neq \ell$ that merges with ℓ at level j. We now fix j and prove that $\mathbf{Pr}[\text{MERGE}_j] \leq 1/4d$. This will finish the proof, since summing over all the levels, we will get that the probability that ℓ is not recoverable is at most 1/4.

We condition on the $r_{j+1}, \ldots, r_{d-1}$, which determines the trajectory of all leaves up to level $j+1$. In particular this determines $i_{j+1}, \ldots, i_d$,

Consider the set of child buckets of $\ell_{\leq j}$. These have the form $(\ell_{\leq j} \circ a, i)$ where $(a,i) \in L \times \{1, \ldots, \kappa_{j+1}\}$. This includes the bucket $(\ell_{\leq j} \circ \ell_{j+1}, i_{j+1})$ on the trajectory of ℓ.

Say that a bucket (v,i) is *occupied* if it lies on the trajectory of some mismatch leaf. Let OCC be the set of pairs $(a,i) \neq (\ell_{j+1}, i_{j+1})$ such that $(\ell_{\leq j} \circ a, i)$ is occupied. The event MERGE_j is equivalent to the event that there is an $(a,i) \in \text{OCC}$ such that $r_j(a,i) = r_j(\ell_{j+1}, i_{j+1})$. For each $(a,i) \in \text{OCC}$, $\mathbf{Pr}[r_j(a,i) = r_j(\ell_{j+1}, i_{j+1})] = \frac{1}{|\kappa_j|}$ since r_j is a pairwise independent map with range size $|\kappa_j|$, and so the conditional probability of MERGE_j given $r_{d-1}, \ldots, r_{j+1}$ is at most $|\text{OCC}|/|\kappa_j|$.

We need to upper bound $|\text{OCC}|$. Let $\mathbf{occ}(v)$ denote the number of occupied v-buckets. In the above analysis $|\text{OCC}| = \sum_{a \in L} \mathbf{occ}(\ell_{\leq j} \circ a) - 1$. We claim:

Proposition 5.8. *For any choice of routing functions $r_{d-1}, \ldots, r_0$:*
(1) For any internal node v at level $j' < d$, $\mathbf{occ}(v) \leq \min(\kappa_{j'}, \sum_{v' \in \mathbf{child}(v)} \mathbf{occ}(v'))$.
(2) For any node v at level $j' \leq d$, $\mathbf{occ}(v) \leq \widehat{\kappa}(v)$.

Proof. For the first part, let v be an internal node. Then $\mathbf{occ}(v)$ is trivially at most $\kappa_{j'}$. Also, a v-bucket is occupied if and only if some occupied child maps to it, so $\mathbf{occ}(v) \leq \sum_{v' \in \mathbf{child}(v)} \mathbf{occ}(v')$.

For the second part, if v is a leaf then $\mathbf{occ}(v) = 1$ if v is a mismatch leaf and 0 otherwise, so $\mathbf{occ}(v) = \widehat{\kappa}(v)$. If v is an internal node, the first part implies $\mathbf{occ}(v) \leq \min(\kappa_{j'}, \sum_{v' \in \mathbf{child}(v)} \mathbf{occ}(v'))$ and

applying induction and the definition of $\widehat{\kappa}(v)$ we have $\min(\kappa_{j'}, \sum_{v' \in \mathbf{child}(v)} \widehat{\kappa}(v')) = \widehat{\kappa}(v)$. □

Thus $|\mathrm{OCC}| \leq \sum_{a \in L} \widehat{\kappa}(\ell_{\leq j} \circ a) - 1$. We know that $\widehat{\kappa}(\ell_{\leq j}) \leq \kappa_j/4d$ and in particular, $\widehat{\kappa}(\ell_{\leq j}) < \kappa_j$. Hence, $\widehat{\kappa}(\ell_{\leq j}) = \sum_{a \in L} \widehat{\kappa}(\ell_{\leq j} \circ a) > |\mathrm{OCC}|$. It follows that for any choice of the routing functions $r_{d-1}, \ldots, r_{j+1}$, $|\mathrm{OCC}| < \kappa_j/4d$. Therefore $\mathbf{Pr}[\mathrm{MERGE}_j] \leq \frac{1}{4d}$, as required to complete the proof of the theorem. □

We are ready to conclude Theorem 5.1. Let us define the sketching function HMR-SKETCH$(u; T(L^d), \kappa, \delta)$ for targeted mismatch recovery $\mathbf{HMR}(T(L^d), \kappa, 4d)$ to be the superposition sketching function on the tree $T(L^d)$ with the capacity function κ, where the distribution on **leaf→root** functions is as given in the above lemma, and with the redundancy set to $\lceil 8(\ln |L^d| + \log(1/\delta) + 2) \rceil$. Let us define HMR-RECOVER to be the associated recovery function. To conclude the correctness of the scheme (the first item of Theorem 5.1) we apply Proposition 5.6 together with Lemma 5.7 with parameters set as follows: $D = L^d$, $S = \{1, \ldots, \kappa_0\}$, redundancy $\ell = \lceil 8(\ln |L^d| + \log(1/\delta) + 2) \rceil$, and (H, μ) as defined above for $(T(L^d), \kappa)$. (Notice, $|\Gamma| \geq 4|L^d|$ implies that $|\Gamma| \geq 4(|D| - 1)| \cdot |S|$ as required by Lemma 5.7.) The sketch consists of $O(\ell \cdot |S|)$ elements from Γ so it takes $O(\kappa_0 \cdot \log |\Gamma| \cdot (\log |L^d| + \log(1/\delta)))$ bits. Evaluating a hash function from H at a single point takes time $O(d \log^{O(1)} |\Gamma|)$ so by Proposition 5.5, the sketching algorithm runs in time $O(\ell \cdot |L^D| \cdot d \log^{O(1)} |\Gamma|) = O(|L^D| \cdot \log^{O(1)} |\Gamma| \cdot \log(1/\delta))$. If u is given via its sparse representation then the time to construct the sketch is $O((\kappa_0 + |\mathbf{supp}(u)|) \cdot \log^{O(1)} |\Gamma| \cdot \log(1/\delta))$. The recovery algorithm runs in time $O(\kappa_0 \cdot \log^{O(1)} |\Gamma| \cdot \log(1/\delta))$ as required. Finally, each mismatch pair that is output by the recovery algorithms must appear in more than half of the ℓ redundant sketches. As each sketch outputs at most $|S|$ elements, the number of mismatch pairs output by the algorithm is at most κ_0.

ACKNOWLEDGMENTS

This work began while the authors were visiting Simons Institute for Theoretical Computer Science, and was greatly advanced while they were at Schloss Dagstuhl–Leibniz Center for Informatics. We are grateful for their generous support.

REFERENCES

[1] Alexandr Andoni and Negev Shekel Nosatzki. 2020. Edit Distance in Near-Linear Time: it's a Constant Factor. *CoRR* abs/2005.07678 (2020). arXiv:2005.07678 https://arxiv.org/abs/2005.07678

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

[3] Djamal Belazzougui and Qin Zhang. 2016. Edit Distance: Sketching, Streaming, and Document Exchange. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 51–60. https://doi.org/10.1109/FOCS.2016.15

[4] Sudatta Bhattacharya and Michal Koucký. 2023. Locally Consistent Decomposition of Strings with Applications to Edit Distance Sketching. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, Barna Saha and Rocco A. Servedio (Eds.). ACM, 219–232. https://doi.org/10.1145/3564246.3585239

[5] Joshua Brakensiek and Aviad Rubinstein. 2020. Constant-factor approximation of near-linear edit distance in near-linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*. ACM, 685–698. https://doi.org/10.1145/3357713.3384282

[6] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. 2020. Approximating Edit Distance Within Constant Factor in Truly Sub-quadratic Time. *J. ACM* 67, 6 (2020), 36:1–36:22. https://doi.org/10.1145/3422823

[7] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. 2016. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, Daniel Wichs and Yishay Mansour (Eds.). ACM, 712–725. https://doi.org/10.1145/2897518.2897577

[8] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. 2009. From coding theory to efficient pattern matching. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*. SIAM, 778–784. https://doi.org/10.1137/1.9781611973068.85

[9] Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. 2019. The streaming k-mismatch problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, Timothy M. Chan (Ed.). SIAM, 1106–1125. https://doi.org/10.1137/1.9781611975482.68

[10] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. 2002. An Approximate L1-Difference Algorithm for Massive Data Streams. *SIAM J. Comput.* 32, 1 (2002), 131–151. https://doi.org/10.1137/S0097539799361701

[11] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. 1992. Nonoblivious Hashing. *J. ACM* 39, 4, 764–782. https://doi.org/10.1145/146585.146591

[12] Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha. 2022. How Compression and Approximation Affect Efficiency in String Distance Measures. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, Joseph (Seffi) Naor and Niv Buchbinder (Eds.). SIAM, 2867–2919. https://doi.org/10.1137/1.9781611977073.112

[13] Szymon Grabowski. 2016. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discret. Appl. Math.* 212 (2016), 96–103. https://doi.org/10.1016/J.DAM.2015.10.040

[14] Ce Jin, Jelani Nelson, and Kewen Wu. 2021. An Improved Sketching Algorithm for Edit Distance. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, (LIPIcs, Vol. 187)*. 45:1–45:16. https://doi.org/10.4230/LIPIcs.STACS.2021.45

[15] Richard M. Karp and Michael O. Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31, 2 (mar 1987), 249–260. https://doi.org/10.1147/rd.312.0249

[16] Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. 2021. Small-space and streaming pattern matching with k edits. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 885–896. https://doi.org/10.1109/FOCS52979.2021.00090

[17] Michal Koucký and Michael E. Saks. 2020. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*. ACM, 699–712. https://doi.org/10.1145/3357713.3384307

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

[19] William J. Masek and Mike Paterson. 1980. A Faster Algorithm Computing String Edit Distances. *J. Comput. Syst. Sci.* 20, 1 (1980), 18–31. https://doi.org/10.1016/0022-0000(80)90002-1

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

[21] Ely Porat and Ohad Lipsky. 2007. Improved Sketching of Hamming Distance with Error Correcting. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4580)*, Bin Ma and Kaizhong Zhang (Eds.). Springer, 173–182. https://doi.org/10.1007/978-3-540-73437-6_19

[22] Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *J. ACM* 21, 1 (1974), 168–173. https://doi.org/10.1145/321796.321811