Simple Low-Overhead Communication-Efficient String Reconciliation and Edit Distance*

Michael T. Goodrich[1][0000−0002−8943−191X], Gonzalo Navarro[2][0000−0002−2286−741X], and Claire A. To[1][0009−0008−9102−2219]

[1] University of California, Irvine, Irvine CA, USA
[2] CeBiB & Dept. of Computer Science, University of Chile, Santiago, Chile

Abstract. Suppose two parties, Alice and Bob, hold long character strings, X and Y, respectively, and they are interested in determining how similar X and Y are. Moreover, they want to exchange the strings with cost proportional to their degree of dissimilarity. In this paper, we provide simple low-overhead communication-efficient algorithms for such string reconciliation and edit distance problems. In the general case, we show how to determine the edit distance k between X and Y using only $O(k^2 \log n)$ bits of communication and optimal $O(n)$ time overhead, with high probability. For specialized cases, such as typical English text or DNA sequences, where we can make additional well-justified assumptions about the distribution of the input strings, we show how to achieve possibly better bounds, such as $O(k \log^5 n)$ bits of communication.

Keywords: distributed algorithms · randomized algorithms · edit distance · hash functions · communication complexity · string reconciliation

1 Introduction

A fundamental challenge in distributed computing lies in managing distributed data across communication networks. A problem arising frequently is that of checking or maintaining consistency across records in replicated databases, synchronizing files in peer-to-peer systems, and repairing or updating databases across the network. Thus, there is a need for efficient distributed protocols for reconciling character strings, which can, e.g., represent DNA sequences, avoiding the communication overhead of retransmitting the whole strings.

Problem Statement. Let $X = x_1 x_2 \dots x_l$ and $Y = y_1 y_2 \dots y_n$ be two strings held by two parties, Alice and Bob, respectively, where each $x_i, y_j \in \Sigma$, for $i = 1, 2, \dots, l$ and $j = 1, 2, \dots, n$, and Σ is an alphabet whose size may or may not depend on l or n. In the case where $l = n$, the *Hamming distance*, $\mathrm{hd}(X, Y)$, between X and Y is the number of indices, i, where $x_i \neq y_i$, that is, the number of character substitutions needed to transform X into Y [8, 24].

* This research supported in part by NSF grant 2212129, by Basal Funds FB0001 and ABF240001, ANID, Chile, and by Fondecyt Grant 1260080, ANID, Chile.

A related notion that does not assume $l = n$ is the ***edit distance***, also known as *Levenshtein distance*, $L(X, Y)$, between X and Y: it is the minimum number character insertions, deletions, or substitutions needed to transform X into Y.

In this paper, we are interested in distributed string reconciliation algorithms with optimal (i.e., linear) computational overhead and small communication complexity for computing the edit distance between the two strings, X and Y, that are respectively held by Alice and Bob. This turns out to be equivalent to the problem of Alice and Bob exchanging their strings X and Y with communication cost based on their communicating a minimal set of edits between X and Y. We assume that Alice and Bob can communicate over some channel but they do not share any other computational resources, such as CPUs or memory. The ***communication complexity*** for a communication protocol for Alice and Bob is the number of bits that are communicated between them over the course of their distributed protocol, where we assume initially that neither party has knowledge of the other party's string.

In the ***general distributed edit distance*** problem, we are interested in a protocol for Alice and Bob to determine the edit distance, $k = L(X, Y)$, between X and Y using a small number of bits of communication (as a function of k). Alternatively, in ***specialized*** string reconciliation problems, we make additional assumptions regarding X and Y and their edit distance. For example, we develop efficient distributed protocols for the case where X and Y come from well-motivated real-world domains, such as English text and DNA sequences, which support assumptions regarding X and Y that we can exploit. Further, for practical reasons, in this paper we are interested in simple algorithms that have low communication complexity and computational overhead for Alice and Bob, for both the general and specialized versions of the string reconciliation problem.

Our results will hold ***with high probability*** (whp), meaning that they occur with probability $1 - 1/n^c$, for some constant $c \geq 1$. W.l.o.g. let us assume $l \leq n$. Our protocols use various independent sources of randomness, e.g., rolling-hash fingerprints, CDC boundary detection, chunk checksums, and IBLT hashing, each of which satisfies the whp guarantee.

Our Contributions. For the general distributed edit distance problem, we present a simple algorithm that achieves a communication complexity of $O(k^2 \log n)$ bits and optimal $O(n)$ time overhead, where k is the edit distance, with a probability of success over $1 - 1/n^2$. Our method is a communication-efficient adaptation of a dynamic-programming table "sliding" algorithm of Landau, Myers, and Schmidt [21] combined with Ukkonen's technique [30]. We avoid performing full sliding steps as in these classical algorithms, however, which would be communication inefficient. Instead, we perform what we call "fast sliding" steps by combining rolling hash functions and noisy binary searching.

We also show how to solve string reconciliation in the specialized context, as described. For such scenarios, we provide an algorithm that uses $O(n)$ time and a communication complexity of

$$O(k(d_X + d_Y)^2 \log n((d_X + d_Y + \log n) \log |\Sigma| + \log \eta \log n))$$

bits, succeeding with probability $1 - 1/\eta$, where k is an upper bound on the edit distance between X and Y known a priori, and d_X and d_Y are the lengths of the longest repeated substrings within X and Y, which are expected to be small for real-world applications and are $O(\log n)$ on widely used statistical models. Under such models, the communication complexity is $O(k \log^5 n)$ bits if $|\Sigma| = O(n^c)$ and the probability of error is $O(1/n^c)$ for any constant c. With $O(n \log k)$ time overhead we obtain the same communication cost, now with k being the actual edit distance, without need to know it a priori. Our method involves a non-trivial use of the invertible Bloom lookup table (IBLT) data structure [10, 12, 15].

2 Related Work

There has been considerable prior work on the general distributed edit distance problem, but we are not aware of any previous methods that are simple and communication-efficient while also having low computational overhead, which refers to the total number of local RAM operations performed by Alice and Bob, excluding time spent transmitting bits over the communication channel. For example, Orlitsky [26] gives a communication-optimal method that achieves $O(k \log n)$ communication complexity, where k is an upper bound on the edit distance between two strings of size $\Theta(n)$, but requires computational overhead that is $n^{O(k)}$, i.e., exponential in k. The computational overhead for such protocols has been subsequently improved [2, 5, 17, 18], but the best methods still require $O(n \operatorname{polylog}(n))$ overhead and are still fairly complicated. For instance, Chakraborty, Goldenberg, and Koucký [5] present a protocol with communication efficiency of $O(k^2 \log n)$ bits and $O(n \operatorname{polylog}(n))$ overhead by embedding strings in a Hamming space using random walks and doing their communication in this Hamming space. Belazzougui and Zhang [3] achieve a communication complexity of $O(k(\log^2 k + \log n))$ bits and $O(n \operatorname{polylog}(n))$ overhead assuming $k < n^{1/c}$ for a large constant c, but their method is quite complex and their bounds have large constant factors. In general, the previous methods for general distributed edit distance [2, 3, 5, 17, 18] all require $O(n \operatorname{polylog}(n))$ overhead, or worse, which in practice is inefficient for the long strings that arise in various real-world settings.

For specialized string reconciliation, Agarwal, Chauhan, and Trachtenberg [1] present distributed methods for reconciling typical real-world strings. Their method has communication complexity that scales linearly in the edit distance between the reconciling strings, as is also true for our methods. Their reconstruction methods are based on using masks and shingling to encode the strings and enumerating Eulerian paths to perform the reconciliation. As they admit, however, the number of such paths (and, hence, the computational burden of such algorithms) can be exponentially large, hence, they do not achieve low overhead. Like our approach, Kontorovich and Trachtenberg [20] reduce string reconciliation to the set reconciliation problem, but their approach still has suboptimal performance. Song and Trachtenberg [28] revisit the approach using masks and shingles with an improved Eulerian-path reconstruction

method, which they show emprically improves over the prior work by Agarwal, Chauhan, and Trachtenberg [1], but the asymptotic overhead of their method is unfortunately still not efficient.

3 A General Algorithm for Edit Distance

In this section, we provide a general communication-efficient distributed algorithm for Alice and Bob to compute the edit distance of their respective strings, $X = x_1x_2 \dots x_l$ and $Y = y_1y_2 \dots y_n$, using $O(k^2 \log n)$ bits of communication and optimal $O(n)$ time overhead, where k is the edit distance between X and Y. Note that in this case we can also assume, without loss of generality, that l is $\Theta(n)$ and $k < \sqrt{n/\log_{|\Sigma|} n}$, since if this is not the case, Alice and Bob can simply exchange X and Y with each other, with $O(n \log |\Sigma|) \subseteq O(k^2 \log n)$ bits of communication, and each can privately compute $L(X, Y)$ via the chosen exact edit-distance algorithm: Our algorithm can start with the $O(k^2 \log n)$-bits plan and switch to this second option right after having communicated $O(n \log |\Sigma|)$ bits.

We first review the relevant hash functions and recall the classic sliding algorithm. We then present our distributed version using $O(k^2 \log^2 n)$ bits of communication, and finally show how to reduce it to $O(k^2 \log n)$.

Hash Functions. In this paper, we assume that hash functions are pseudo-random and can be considered as random functions for the sake of analysis. One particular type of hash function we use is a ***rolling hash function***, which can be computed for substrings of a string, $X = x_1x_2 \dots x_l$. In particular, the well-known Rabin-Karp string pattern matching algorithm [19] uses the following rolling hash function, for hashing the substring, $x_ix_{i+1} \dots x_j$, in which we view each x_i as an integer:

$$\mathrm{rh}(x_ix_{i+1} \dots x_j) = x_ib^{j-i} + x_{i+1}b^{j-i-1} + \dots + x_jb^0.$$

Here, all arithmetic is done on a suitably chosen finite field (e.g., modulo some prime p) and b is a randomly chosen generator for that field. We also use this rolling hash function, but rather than using it only for fixed-length substrings, as in the Rabin-Karp algorithm, we use it in some cases for fixed-length substrings and other times for arbitrarily long substrings. One useful property of this rolling hash function is that we can compute the rolling hash for every prefix of a length-l string, X, in $O(l)$ time using the following inductive formula for each prefix:

$$\mathrm{rh}(x_1x_2 \dots x_i) = \mathrm{rh}(x_1x_2 \dots x_{i-1})b + x_i.$$

Then, if we store these hash values for every prefix of X, we can compute the value of $\mathrm{rh}(x_ix_{i+1} \dots x_j)$ for any substring, $x_ix_{i+1} \dots x_j$, in constant time as follows:

$$\mathrm{rh}(x_ix_{i+1} \dots x_j) = \mathrm{rh}(x_1x_2 \dots x_j) - \mathrm{rh}(x_1x_2 \dots x_{i-1})b^{j-i+1}.$$

Alternatively, if we are interested in computing rolling hash values of substrings that have a fixed-length, r, then we can use the following formula to compute the next hash value from the previous one in constant time:

$$\mathrm{rh}(x_i x_{i+1} \ldots x_{i+r-1}) = \mathrm{rh}(x_{i-1} x_i \ldots x_{i+r-2})b + x_{i+r-1} - x_{i-1}b^r.$$

The Classic Sliding Algorithm. We first review a classic "sliding" algorithm to compute edit distance by Landau, Myers, and Schmidt [21], combined with Ukkonen's technique [30]. Let us begin describing the classic dynamic programming algorithm for computing edit distance [33]. Let $L(i, j)$ denote the edit distance for the prefixes $X_i = x_1 x_2 \ldots x_i$ of X and $Y_j = y_1 y_2 \ldots y_j$ of Y. Then, we have the boundary cases $L(i, 0) = i$ and $L(0, j) = j$, and, in general, the recurrence:

$$L(i,j) = \min \begin{cases} L(i-1,j)+1 & \text{(deletion)} \\ L(i,j-1)+1 & \text{(insertion)} \\ L(i-1,j-1)+\delta(x_i,y_j), & \end{cases}$$

where $\delta(x_i, y_j) = 0$ if $x_i = y_j$ (match) and $\delta(x_i, y_j) = 1$ if $x_i \neq y_j$ (substitution). This set of equations defines an $l \times n$ matrix, L, where $L(l, n)$ is the edit distance between X and Y, and the sequence of values that lead to this value can be viewed as a path that starts at $L(0, 0)$ and follows horizontally rightward, vertically downward, or down-right diagonal edges between adjacent cells.

If $L(l, n) \leq k$, this path will be restricted to fall between the $j = i - k$ and $j = i + k$ diagonals of $L(i, j)$ entries [31], which we call the ***central band*** of L.

As noted by Landau *et al.* [21], most of the edges in the central band of L go between cells that have the same values and only $O(k^2)$ of these go between cells whose values differ, and these differ by 1. The key idea in their algorithm is to use suffix trees [22, 34] to quickly "slide" along diagonal paths where values in L are not changing.

For a diagonal, d, and row, i, let $\mathrm{Slide}(d, i)$ denote the furthest row, $\rho \geq i$, that can be reached from row i on diagonal d with edit cost 0. That is,

$$\mathrm{Slide}(d,i) = \max\{\rho \mid i \leq \rho \leq \min\{l, n-d\} \\ \text{and } x_{i+1} \ldots x_\rho = y_{i+d+1} \ldots y_{\rho+d}\}.$$

Note that $\mathrm{Slide}(d, i)$ does not compare x_i and y_i. Also, note that $\mathrm{Slide}(0, 0)$ is the length of the longest common prefix of X and Y. Further, if $x_{i+1} \neq y_{i+d+1}$, then $\mathrm{Slide}(d, i) = i$, since the matching substrings of X and Y in the definition of $\mathrm{Slide}(d, i)$ in this case are empty.

In addition, define $M^h(d)$ to be the maximum row index of an entry on diagonal d that has value h, and define the ***h-wave*** to be the set

$$M^h = \{M^h(-h), M^h(-h+1), \ldots, M^h(h-1), M^h(h)\}.$$

To handle boundary cases, we set $M^h(d) = \infty$ if $M^h(d)$ does not actually exist.

At a high level, the algorithm computes the h-waves, for $h = 0, 1, 2, \ldots$, until a wave h' is computed for which $M^{h'}(n-l) = l$, in which case the algorithm outputs h' as the edit distance. The "inner loop" formula for computing the h-wave from the $(h-1)$-wave, for $h > 0$, is the following [21]:

$$M^h(d) = \text{Slide}(d, i_{\max}(d, h)),$$

where

$$i_{\max}(d,h) = \max \begin{cases} M^{h-1}(d+1)+1 & \text{if } d < h-1 \\ M^{h-1}(d)+1 & \text{if } -h < d < h \\ M^{h-1}(d-1) & \text{if } d > -h+1. \end{cases}$$

Since there are only $O(k)$ entries from the matrix L in each h-wave that we need to consider and only k h-waves that we need to consider, this implies that we just need to perform $O(k^2)$ slide computations to determine the edit distance.

Because k is unknown in advance, we compute the waves incrementally. Following Ukkonen [30], we first compute the 0-wave, for which we need only the diagonal 0; we then compute the 1-wave, for which we need only the diagonals -1, 0, and 1. In general, computing the h-wave only requires diagonals $-h$ to h. Thus, for the k-th wave, we compute

$$\sum_{h=0}^{k} (2h+1) = O(k^2)$$

wave entries $M^h(d)$ in order to find the edit distance $k = h'$. By computing each entry in $O(1)$ time using the suffix trees of X and Y, it is possible to compute the edit distance in time $O(k^2)$.

Our Fast-Sliding Algorithm. Our algorithm, which we call the "fast-sliding algorithm," is a communication-efficient adaptation of the sliding algorithm; see also, e.g., [5]. In the distributed setting, we cannot afford to use suffix trees as done by Landau *et al.*, as exchanging them would be too costly. Our method instead uses a Karp–Rabin rolling hash function, rh. In particular, to compute $\text{Slide}(d, i)$, in the general case, we need to find the maximum matching index, ρ, such that

$$x_{i+1} \ldots x_\rho = y_{i+d+1} \ldots y_{\rho+d}.$$

We do this by a doubling-binary search [4] starting from the index $i+1$ in X and index $i+d+1$ in Y, where we communicate and compare $\text{rh}(x_{i+1} \ldots x_t)$ and $\text{rh}(y_{i+d+1} \ldots y_{t+d})$, for $t > i+1$.

We choose the parameters for rh in this case so that the hash values have at least $3 \log n$ bits, so that with probability at least $1 - n^{-3}$ each test during a doubling-binary search will successfully determine whether $x_{i+1} \ldots x_t = y_{i+d+1} \ldots y_{t+d}$ just by comparing the respective rolling-hash values.

Lemma 1. *The fast-sliding algorithm uses $O(k^2 \log^2 n)$ bits of communication and succeeds with high probability.*

Proof: Each doubling-binary search requires $O(\log n)$ such tests, each communicating $O(\log n)$ bits using the rolling hash. Since the classic sliding algorithm performs $O(k^2)$ slide computations, this implies that the entire algorithm has a communication complexity of $O(k^2 \log^2 n)$ bits and performs $O(k^2 \log n)$ hash comparisons. Under our assumption that $k < \sqrt{n/\log_{|\Sigma|} n}$, the number of hash comparisons is $O(n)$. Each hash comparison fails with probability at most n^{-3}, so by a union bound over all comparisons gives an overall failure probability of $O(n^{-2})$. ■

Improving the Communication Complexity. We can reduce the communication per comparison to $O(1)$ bits by treating the comparisons during a doubling-binary search as "noisy" binary-search queries, similar to a technique used by Viola [32]. We first compute an $O(\log n)$-bit rolling hash

$$rh(\cdot) = rh_1(\cdot), \ldots, rh_{L_c}(\cdot)$$

where each rh_j is a 4-bit hash component and $L_c \approx 6 \log_2 n$. For the j-th comparison, Alice and Bob compare only the j-th 4-bit component.

If two substrings are different, the probability a single 4-bit component matches is $2^{-4} = 1/16$, so each comparison succeeds with probability at least $15/16$, and the result of each comparison is independent of any other. We use the noisy-search procedure of [9], also see e.g., [11, 13].

Lemma 2. *For a noisy comparison with error probability $p = 1/16$, the noisy-search procedure of [9] finds the target among n indices with failure probability $\delta = n^{-3}$ using $O(\log n)$ queries, each requiring $O(1)$ bits of communication.*

Proof: By the noisy-search bound from [9], the expected number of queries is upper bounded by

$$\frac{(\log_2 n + \log_2(\delta^{-1}) + 3)}{I(p)}$$

where $I(p) = 1 - H(p)$ are the bits of information gained and $H(p) = -p \log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function. For $\delta = n^{-3}$ and $p = 1/16$, we have an upper bound of $\approx 6.035 \log_2 n + 4.527$ queries, each exchanging a 4-bit component. ■

Thus, we have the following.

Theorem 1 *Let X and Y be strings of length l and n, respectively, with $l \leq n$, from a finite alphabet, stored with separate parties. Then we can compute the edit distance k between X and Y with an overhead of $O(n)$ computational operations for the two parties and a communication complexity of $O(k^2 \log n)$ bits whp.*

Proof: The fast-sliding algorithm performs $O(k^2)$ doubling-binary searches, and by Lemma 2, each search requires $O(\log n)$ bits of communication and succeeds whp. The total local computation consists of constructing the rolling hash in

$O(n)$ and performing $O(k^2 \log n) = O(n)$ hashes, since $k < \sqrt{n/\log_{|\Sigma|} n}$. Thus, the total communication is $O(k^2 \log n)$ bits with $O(n)$ overhead. ■

Note that Alice and Bob not only discover the edit distance; they can reconstruct the path in their matrix L that leads to such distance. This allows each party reconstruct the other string, with $O(n)$ additional time and space [23].

4 A Specialized Algorithm based on IBLT Chunking

In many applications, the strings stored by Alice and Bob come from input distributions that follow widely accepted statistical models [7, 27], as is the case of DNA and English text. Those models imply that all strings over some short length are unique whp.[3] In such cases, and if we know a priori an upper bound k on the edit distance, we are able to possibly improve the communication complexity of the fast-sliding algorithm of the previous section, to $O(k \log^5 n)$ bits. Concretely, we devise an efficient string reconciliation algorithm that is based on ***content-defined chunking*** [6, 35]. We refer to this as our ***IBLT-chunking*** algorithm, since it also uses an invertible Bloom lookup table (IBLT) [15]. This data structure is also used by Eppstein *et al.* [12] to efficiently compute Hamming distance, but our usage is different. We first give some background on IBLTs and then describe our algorithm.

Invertible Bloom Lookup Table (IBLT). IBLT [10, 15] is a space-efficient probabilistic data structure, based on Bloom filters, that supports various operations implemented by the XOR primitive. It can be used to solve the *set reconciliation* problem [12], which is the task of synchronizing two sets stored at two nodes across a communication link. IBLTs allow the parties to efficiently identify the items that are unique to each set so that both parties can update their local copies and obtain the other set as well. The main step in this process is to compute a *set difference*, finding the set of elements that one party possesses but the other does not.

Suppose we are given a set $S = \{s_1, s_2, \ldots, s_n\}$, which we want to store probabilistically in an IBLT table, T, with m cells. We assume that we have an ***encoding function***, `encode`, which encodes each element, s_i, into an associated unique, fixed-length binary string key, e_i. We also have λ hash functions that map any key to λ distinct locations, and we let HashToIndices(e_i, λ, m) denote the set of λ locations determined by these hash functions for the key e_i. Each IBLT cell, $T[j]$, stores a `(keysum, hashsum)` pair such that `keysum` contains the bitwise-XOR (denoted $\oplus$) of all encoded keys e_i that are mapped to $T[j]$ by one of T's hash functions; `hashsum` contains the XOR of all fingerprints or hashes $H(e_i)$, where $H(e_i)$ is an ℓ-bit secondary hash of e_i, such as a checksum, or cryptographic hash, used to verify or detect errors in the decoding process.

[3] Note that this is not the case in some applications, for example those where repetitive sequences are handled [25].

Algorithm 1 IBLT Subtract ($T = T_A \oplus T_B$)

```
for i = 0 to m − 1 do
    T[i].keysum ← T_A[i].keysum ⊕ T_B[i].keysum
    T[i].hashsum ← T_A[i].hashsum ⊕ T_B[i].hashsum
```

Algorithm 2 IBLT Decode T

```
while ∃ i, s.t. H(T[i].keysum) = T[i].hashsum do
    s ← decode(T[i].keysum)
    Output s
    delete(T, s)
for i = 0 to m − 1 do
    if T[j].keysum ≠ 0 OR T[j].hashsum ≠ 0 then
        return FAIL
return SUCCESS
```

The IBLT supports both insertion and deletion of elements using XOR operations. Inserting an element corresponds to XOR-ing its encoded value into the table (at the λ locations), while deleting it corresponds to *the same* operations.[4] To encode an entire set, S, we simply insert each of its elements.

For a set reconciliation protocol, Alice and Bob build respective tables, T_A and T_B, by inserting all the elements of their sets, X and Y, respectively, into initially zeroed IBLTs of the same size using the same hash functions, and exchange the tables. From these two IBLTs, T_A and T_B, both Alice and Bob can compute an IBLT representing the symmetric difference between X and Y. This simple method is shown in Algorithm 1. The fact that this algorithm computes an IBLT representation of the symmetric difference of X and Y follows immediately from the fact that $x \oplus x = 0$ for any value x.

Listing Set Entries If an IBLT is not "too" full, we can list out its entries. The listing of entries or decoding is a destructive $O(m)$-time procedure that recovers all keys in the IBLT or reports that the IBLT is too full to achieve this successfully. Say that a cell $T[i]$ is ***pure*** if

$$H(T[i].\texttt{keysum}) = T[i].\texttt{hashsum},$$

which clearly holds for a cell holding exactly one key but could admittedly also hold if there is a checksum collision. Note that if a cell is pure, we can recover its key by extracting the value from $T[i].\texttt{keysum}$ and then deleting the element from the table, which is also referred to as ***peeling*** they key from the corresponding λ hashed cells. This may make other cells pure, which can eventually allow us to recover all the values stored in the IBLT. Algorithm 2 shows this process.

[4] This method assumes each item is inserted or deleted at most once, which is true for sets.

Lemma 3. *The probability that a cell containing $t \geq 2$ keys produces a false positive on the purity test is $2^{-\ell}$.*

Proof: Let H be a uniform random hash function that produces ℓ-bit hash values. Suppose a cell contains $t \geq 2$ keys, with $\texttt{keysum} = x_1 \oplus x_2 \oplus \ldots \oplus x_t$ and $\texttt{hashsum} = H(x_1) \oplus H(x_2) \oplus \ldots \oplus H(x_t)$. Define a random variable $Z = H(\texttt{keysum}) \oplus \texttt{hashsum} = H(x_1 \oplus \ldots \oplus x_t) \oplus H(x_1) \oplus \ldots \oplus H(x_t)$. Because H is uniform and random, $H(x_1 \oplus \cdots \oplus x_t)$ is independent of $H(x_1), \ldots, H(x_t)$ and each value is uniformly distributed over $\{0,1\}^\ell$. Thus, the probability of failure is $2^{-\ell}$. ■

Corollary 1. *In an IBLT of m cells, the probability that the purity test yields a false positive for any cell is at most $m2^{-\ell}$.*

Proof: Apply the union bound to the failure probability from Lemma 3. ■

In our setting, we want to recover a symmetric difference of size at most d with high probability, even if d is a constant. We use an IBLT of size $m = c \cdot d \log n$ for a $c \geq 1$, and λ being $\Theta(\log n)$, so that by an analysis similar to that of Goodrich, Kitagawa, and Mitzenmacher [14], when the current number of elements in the IBLT is at most d, we can successfully list all entries of T whp:

Theorem 2 *Let T be an IBLT with $m = cd \log n$ cells and $\lambda = (c/2) \log n$ hash functions, for $c \geq 4$. If T stores at most d elements, then it can be successfully decoded by the peeling algorithm with probability at least $1 - 1/n^{c/2-1}$.*

Proof: Let X_i be a random variable that is 1 if the i-th element has no pure cell and is 0 otherwise. Since there are at most d elements in T and it has $m = cd \log n$ cells and $\lambda = (c/2) \log n$ hash functions, at least half the cells in the IBLT are empty. Thus, any one of the λ hash functions will map an element to what would otherwise be an empty cell with probability at least $1/2$. Since the λ hash functions are independent, this implies that

$$\Pr(X_i = 1) \leq \frac{1}{2^\lambda} = \frac{1}{n^{c/2}}.$$

Thus, by a union bound, the probability that every element in T has a pure cell is at least $1 - 1/n^{c/2-1}$. ■

To sum up, suppose two sets, X and Y, of size $O(n)$, are held by Alice and Bob respectively, which are assumed to differ by at most d elements. Each party independently constructs an IBLT, T_A and T_B of $m = O(d \log n)$ cells, and inserts all elements into the table. Even if either T_A and T_B is too full to decode individually, there will be pure cells in $T_A \oplus T_B$, i.e., containing only one key. That is, with properly chosen parameters, Alice and Bob can reconcile their sets whp, in $O(n)$ time and $O(d\beta \log n)$ bits of communication, where $\beta > 0$ is the size of the keys, by exchanging their tables T_A and T_B and performing the decoding algorithm on $T_A \oplus T_B$.

Our Algorithm. Content-defined chunking (CDC) [6, 35] divides a string, $X = x_1x_2 \ldots x_n$, into a set, $\mathcal{C}$, of contiguous substrings, called ***chunks***, X_1, X_2, ..., X_r, so that $X = X_1||X_2|| \cdots ||X_r$, where $||$ denotes concatenation. This is achieved by defining a minimum-size parameter, s, and a windowed rolling hash function, h, with a window size, w, and subdividing the string, X, by a simple algorithm that computes h for each substring of X of length w, starting from the beginning of X. At each position where $h(x_i \ldots x_{i+w-1}) = 0$, if the length of the current substring being processed is at least s, then we cut X at the index $i + w - 1$ to form the next chunk.

In the edit distance algorithm of this section, we desire that the chunks in $\mathcal{C}$ are unique and not too long, which we can characterize in terms of a parameter d_X: the length of the longest repeated string in X. Note that d_X is computed in linear time using suffix trees. The following observation should be obvious.

Lemma 4. *Let $\mathcal{C} = \{X_1, X_2, \ldots, X_r\}$ be a set of chunks for the string $X = X_1||X_2|| \cdots ||X_r$. If $|X_i| > d_X$ for $i = 1, 2, \ldots, r$, then each chunk in $\mathcal{C}$ is unique.*

Szpankowski [29] shows that the value of d_X for strings X that can be modeled as pseudo-random sequences, such as those determined by Bernoulli or Markov processes, is $O(\log |X|)$ whp. Note that d_X is always at least $\log_{|\Sigma|} |X|$. Appendix A of the extended version [16] discusses practical values.

Before running the string reconciliation protocol, Alice and Bob first compute and exchange d_X and d_Y, so that both use the same minimum chunk size, $s = \max\{d_X, d_Y\} + 1$, and the same CDC window size, $w = s$, for their respective chunking processes. Thus, no chunk boundary is allowed before it has accumulated s characters. For the IBLT construction, we choose a confidence bound, $n < \eta \leq n^c$, for some constant $c \geq 1$, where we desire our protocol to succeed with probability at least $1 - 1/\eta$ (i.e., whp).

Lemma 5. *With high probability, the maximum chunk size in $\mathcal{C}(X)$ (resp., $\mathcal{C}(Y)$) is $O(s + \log |X|)$ (resp., $O(s + \log |Y|)$).*

Proof: W.l.o.g., let us focus on Alice's set of chunks, $\mathcal{C}(X)$. By definition, every substring of X with length at least s is guaranteed to be unique. Suppose that the uniformly random rolling hash function h outputs b bits. The probability of a cut at any window w is $p = Pr[h(x_i \ldots x_{i+w-1}) = 0] = 1/2^b$. Suppose a chunk of length $l > w + t$ starts at position i. Then the chunk contains at least t distinct windows of size w, with starting positions in the range $[i, i + t - 1]$. For the chunk to have length greater than $w + t$, none of these t windows can produce a cut. Therefore, $\Pr[l > w + t] \leq (1 - p)^t \leq e^{-pt}$. Let $t = (c/p) \log |X|$ for a constant $c \geq 2$, then $Pr[l > w + t] \leq e^{-pt} = e^{-c \log |X|} = 1/|X|^c$. By a union bound over all $|X|$ possible starting positions, the probability any chunk exceeds $w + t$ is at most $1/|X|^{c-1}$. Since $w = s$ and $t = O(\log |X|)$ for constant hash output size b (which makes p constant), with high success probability, at least $1 - 1/|X|^{c-1}$ where $c \geq 2$, the maximum chunk size is $O(s + \log |X|)$. ■

Let us further assume that we have an upper-bound estimate, $k \geq 1$, for the edit distance between the strings, X and Y, held respectively by Alice and

Bob (we later lift this assumption). Then we create an IBLT of appropriate size to allow listing its elements with probability at least $1 - 1/\eta$, for our desired confidence parameter of successful decoding, $n < \eta \leq n^c$, if there are most κ elements, i.e., an IBLT of $O(\kappa \log \eta)$ cells, with each cell storing a `keysum` and `hashsum` of $\Theta(\log n)$ bits (here, κ is a function of k to be defined later). Alice then stores a suitably defined set, $S(X)$, in her IBLT of this chosen size such that, given a string, X, and a set of chunks, $\mathcal{C}(X) = \{X_1, X_2, \ldots, X_r\}$, for $X = X_1||X_2|| \cdots ||X_r$, her IBLT is defined with respect to the set $S(X)$ defined for her chunks. In particular, she encodes $X = x_1 x_2 \ldots x_n$ as a set of pairs of checksum hashes of consecutive chunks,

$$S(X) = \{H(X_i)||H(X_{i+1}) \mid X_i \in \mathcal{C}(X), i = 1, \ldots, r-1\},$$

where $H(X')$ is a checksum hash with $\Theta(\log n)$ bits for the chunk X', so that, whp, all the checksum hashes are unique, given that all the chunks are unique. Thus, each key associated with consecutive chunks is the concatenation of their fixed-length hash values and is itself a fixed-length bit string. Likewise, Bob encodes a similar set of pairs of chunk checksums, $S(Y)$, and stores these in a similarly-defined IBLT of the same size. Alice and Bob then exchange their IBLTs and perform the peeling algorithm to find the set difference.

For each differing chunk pair that either Alice or Bob decodes, they can determine if it is one of their chunk pairs or a chunk pair from the other party. Further, after Alice and Bob have respectively decoded the set-difference IBLT and determined which chunk pairs belong to X and which belong to Y, Alice and Bob then each send the other their corresponding chunk pairs. That is, for each chunk-checksum pair, $H(X_i)||H(X_{i+1})$, of hers that Alice decodes from the IBLT as being a part of the symmetric difference, Alice sends Bob the pair (X_i, X_{i+1}). Likewise, Bob sends a similar set of chunk pairs to Alice.

Lemma 6. *Let k be an upper bound on the edit distance between two strings, X and Y, each of length $\Theta(n)$, defined over an alphabet Σ, let d_X and d_Y be the length of their longest repeated substring, respectively, and $s = \max\{d_X, d_Y\} + 1$. Then our IBLT-chunking algorithm finds the set of $O(k)$ differing chunk pairs between X and Y with probability at least $1 - 1/\eta$, for a desired confidence bound, $n < \eta \leq n^c$, for a constant $c \geq 1$, using $O(ks^2 \log n((s + \log n) \log |\Sigma| + \log \eta \log n))$ bits of communication.*

Proof: By the above discussion and Lemma 5, every chunk has length $O(s + \log n)$ characters whp. Encoding each character requires $\lceil \log |\Sigma| \rceil$ bits, so each chunk will have size $O((s + \log n) \log |\Sigma|)$ bits whp. We show that the number of differing chunks to exchange is $\kappa = O(ks^2 \log n)$ whp. Plus, the IBLT needs $O(\kappa \log \eta)$ cells of $O(\log n)$ bits each to decode with probability at least $1 - 1/\eta$.

Since any edit between X and Y will change at most w consecutive rolling hashes, and since $s = w$, those positions span at most two consecutive chunks. The problem is that, because of the edit, the rolling hashes on X and Y may yield zero at different positions, and since we do not consider other zeros before scanning the next $s = w$ symbols, the next zeros considered on X and Y

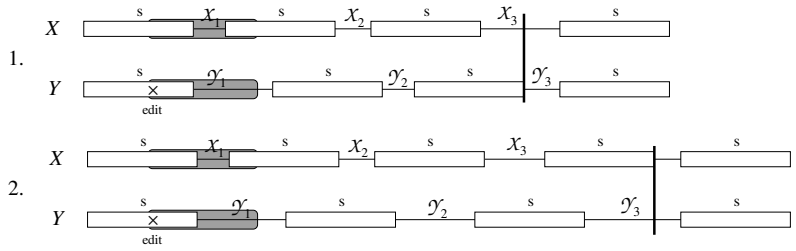

Fig. 1. Cases of resynchronization. Chunks are a rectangle (where they cannot end) followed by a line (of lengths $\mathcal{X}_i$ or $\mathcal{Y}_i$) where they end once the rolling hash is 0. Hashes can differ in the grayed areas only. Thick vertical lines show the synchronization point.

may differ again, even if the preceding windows are identical by then. This propagation, however, ends soon whp: Assume both processes are synchronized up to now. Once an edit occurs, and after passing the minimum s symbols if required, we can take the distance to the next zero of the rolling hash in X as a random variable $\mathcal{X}_1$, and in Y as a random variable $\mathcal{Y}_1$, both geometrically distributed with mean 2^b. If $\mathcal{X}_1 = \mathcal{Y}_1$, then they have resynchronized. Otherwise, assume w.l.o.g. $\mathcal{X}_1 < \mathcal{Y}_1$, and call $\mathcal{X}_i$ and $\mathcal{Y}_i$ the number of rolling hash probes in the ith chunk. The process can resynchronize in two ways (see Figure 1):

1. For some $\tau \geq 1$, $\mathcal{X}_1 + s + \cdots + \mathcal{X}_\tau + s + \mathcal{X}_{\tau+1} \geq \mathcal{Y}_1 + s + \cdots + \mathcal{Y}_\tau + s = j$.
2. For some $\tau \geq 1$, $j = \mathcal{X}_1 + s + \cdots + \mathcal{X}_\tau + s \leq \mathcal{Y}_1 + s + \cdots + \mathcal{Y}_\tau$.

In both cases, the two windows ending at position j are identical (far away from the edit position) and of length at least s, so both chunks are ended at j. Let $\mathcal{Z}_k = \sum_{i=1}^{k} \mathcal{X}_i - \mathcal{Y}_i$: we resynchronize as soon as (1) $\mathcal{Z}_\tau + \mathcal{X}_{\tau+1} \geq 0$ or (2) $\mathcal{Z}_\tau \leq -s$ for some τ. In Appendix B of the full version [16] we show that this occurs whp after $\Theta((s^2/4^b)\log n)$ steps, and on average after $\Theta(s^2/4^b)$ steps. For insertions or deletions, the positions are shifted by ± 1 but the result is identical.

Thus, if k is an upper bound on the edit distance between X and Y, then at most $\kappa = O(ks^2 \log n)$ chunks differ, whp, between $\mathcal{C}(X)$ and $\mathcal{C}(Y)$; hence, $O(ks^2 \log n)$ chunk-checksum pairs are different between $S(X)$ and $S(Y)$ whp. Thus, after the decoding of the IBLT occurs, Alice and Bob each send the other lists of $O(ks^2 \log n)$ chunk pairs, each of at most $O((s + \log n)\log|\Sigma|)$ bits. ■

In addition, the information exchanged is sufficient for each party to reconstruct the other party's string. W.l.o.g., let us focus on Bob, who wishes to reconstruct X by "linking" all of Alice's chunk pairs. Also, with probability at least $1 - 1/\eta$, we can assume all the elements in the symmetric difference were decoded correctly and that all the chunk checksums are unique. For instance, the chunks in $\mathcal{C}(X)$ (resp., $\mathcal{C}(Y)$) are unique by construction; hence, the chunk-checksum pairs in $S(X)$ and $S(Y)$ are also respectively unique whp. Further, assume for simplicity that Alice always sends Bob her first chunk pair (X_1, X_2), so Bob can determine if their first chunk pairs match or not. In either case, he can determine the first two chunks in $\mathcal{C}(X)$. Also, note that Bob has received

from Alice each chunk pair, (X_j, X_{j+1}), such that $H(X_j)||H(X_{j+1})$ is a chunk-checksum pair in the symmetric difference of $S(X)$ and $S(Y)$ decoded from the IBLT. Suppose (X_{i-1}, X_i) is the last pair of chunks in $\mathcal{C}(X)$ that Bob has discovered; hence, he is interested in discovering the next pair, (X_i, X_{i+1}).

- Case 1. There is a chunk pair, $H(X_i)||H(X')$, in the IBLT symmetric difference. Since chunks are unique, this implies $(X_i, X_{i+1}) = (X_i, X')$. Moreover, in this case, Alice has sent Bob the pair, (X_i, X'); hence, Bob can determine (X_i, X_{i+1}).
- Case 2. There is no chunk pair, $H(X_i)||H(X')$, in the IBLT symmetric difference. Then there is a chunk-checksum pair, $H(X_i)||H(X')$, that is also in $S(Y)$; i.e., $(X_i, X') = (Y_j, Y_{j+1})$, for some $j > 1$. Since chunks are unique, this implies $(X_i, X_{i+1}) = (Y_j, Y_{j+1})$; hence, Bob can determine (X_i, X_{i+1}) by looking up his chunk, Y_{j+1}, given that he has already determined that X_i matches Y_j.

Corollary 2. *The exchanged information is sufficient for Bob (resp., Alice) to reconstruct X (resp., Y) with probability at least $1 - 1/\eta$.*

Therefore, we have the following result, which yields $O(k \log^5 n)$-bit communication complexity whp in the discussed models where d_X and d_Y are $O(\log n)$.

Theorem 3 *Let k be an a-priori-known upper bound on the edit distance between two strings, X and Y, of length $\Theta(n)$, each defined over an alphabet Σ, held by two separate parties, Alice and Bob, let d_X and d_Y be the lengths of their longest repeated substrings, and $s = \max\{d_X, d_Y\} + 1$. Then our IBLT-chunking algorithm allows Alice and Bob to determine the other party's string in $O(n)$ time using*

$$O(ks^2 \log n(s \log |\Sigma| + \log \eta \log n))$$

bits of communication with probability at least $1 - 1/\eta$.

Once both Alice and Bob have the chunks of both strings, they can compute their edit distance in $O(n + k^2)$ additional time [21].

Thus, when $k = O(\sqrt{n})$ and d_X and d_Y are $O(\log n)$, we can also compute the precise edit distance within this cost, which may outperform our general fast-sliding algorithm.

To conclude, we note that we can obtain a similar result *without* knowing an *a priori* bound k on the edit distance, by slightly increasing the time overhead.

Corollary 3. *The same result of Theorem 3 can be obtained if k is the actual edit distance between X and Y, a priori unknown, with $O(n \log k)$ time overhead.*

Proof: Apply Theorem 3 with upper bound $k' = 1, 2, 4, 8, \ldots$ until it succeeds. With probability at least $1 - 1/\eta$, the try with $k \leq k' < 2k$ will succeed, and each party will obtain the other's string. The total communication cost is as in Theorem 3, replacing k by $\sum_{j=0}^{\lceil \log_2 k \rceil} 2^j = O(k)$, and the time overhead is $O(n \log k)$ because we process the chunks $O(\log k)$ times, once per value of k'. ■

References

1. S. Agarwal, V. Chauhan, and A. Trachtenberg, "Bandwidth efficient string reconciliation using puzzles," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1217–1225, 2006.
2. D. Belazzougui, "Efficient deterministic single round document exchange for edit distance," *CoRR*, vol. 1511.09229, 2015.
3. D. Belazzougui and Q. Zhang, "Edit distance: Sketching, streaming, and document exchange," in *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, 2016, pp. 51–60.
4. J. L. Bentley and A. C.-C. Yao, "An almost optimal algorithm for unbounded searching," *Information Processing Letters*, vol. 5, no. 3, pp. 82–87, 1976.
5. D. Chakraborty, E. Goldenberg, and M. Koucký, "Streaming algorithms for embedding and computing edit distance in the low distance regime," in *Proc. ACM Symp. on Theory of Computing (STOC)*, 2016, pp. 712–725.
6. B. Chapuis, B. Garbinato, and P. Andritsos, "Throughput: A key performance measure of content-defined chunking algorithms," in *Proc. 36th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, 2016, pp. 7–12.
7. T. Cover and J. Thomas, *Elements of Information Theory*, 2nd ed. Wiley, 2006.
8. M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on Strings*. Cambridge University Press, 2007.
9. D. Dereniowski, A. Łukasiewicz, and P. Uznański, "Noisy (binary) searching: Simple, fast and correct," in *Proc. 42nd Symp. on Theoretical Aspects of Computer Science (STACS)*, 2025, pp. 29:1–29:18.
10. D. Eppstein and M. T. Goodrich, "Straggler identification in round-trip data streams via newton's identities and invertible Bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 2, pp. 297–306, 2010.
11. D. Eppstein, M. T. Goodrich, and V. Sridhar, "Computational geometry with probabilistically noisy primitive operations," *CoRR*, vol. 2501.07707, 2025.
12. D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," *SIGCOMM Computer Communication Review*, vol. 41, no. 4, p. 218–229, Aug. 2011.
13. U. Feige, P. Raghavan, D. Peleg, and E. Upfal, "Computing with noisy information," *SIAM Journal on Computing*, vol. 23, no. 5, pp. 1001–1018, 1994.
14. M. T. Goodrich, R. Kitagawa, and M. Mitzenmacher, "Parallel peeling of invertible Bloom lookup tables in a constant number of rounds," in *Proc. Int. Conf. on Current Trends in Theory and Practice of Computer Science*, 2025, pp. 70–84.
15. M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," *CoRR*, vol. 1101.2245, 2015.
16. M. T. Goordrich, G. Navarro, and C. A. To, "Simple low-overhead communication-efficient string reconciliation and edit distance," *CoRR*, vol. 2608.19149, 2026.
17. U. Irmak, S. Mihaylov, and T. Suel, "Improved single-round protocols for remote file synchronization," in *Proc. 24th Joint Conf. of the IEEE Computer and Communications Societies*, vol. 3, 2005, pp. 1665–1676.
18. H. Jowhari, "Efficient communication protocols for deciding edit distance," in *Proc. European Symp. on Algorithms (ESA)*, 2012, pp. 648–658.
19. R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
20. A. Kontorovich and A. Trachtenberg, "String reconciliation with unknown edit distance," in *Proc. IEEE Int. Symp. on Information Theory*, 2012, pp. 2751–2755.

21. G. M. Landau, E. W. Myers, and J. P. Schmidt, "Incremental string comparison," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 557–582, 1998.
22. E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.
23. E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.
24. G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.
25. ——, "Indexing highly repetitive string collections, part I: Repetitiveness measures," *ACM Computing Surveys*, vol. 54, no. 2, p. article 29, 2021.
26. A. Orlitsky, "Interactive communication: Balanced distributions, correlated files, and average-case complexity," in *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, 1991, pp. 228–238.
27. C. E. Shannon, "A mathematical theory of communication," *Bell Systems Technical Journal*, vol. 27, pp. 398–403, 1948.
28. B. Song and A. Trachtenberg, "Scalable string reconciliation by recursive content-dependent shingling," in *Proc. 57th Allerton Conf. on Communication, Control, and Computing (Allerton)*, 2019, pp. 623–630.
29. W. Szpankowski, "A generalized suffix tree and its (un)expected asymptotic behaviors," *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1176–1198, 1993.
30. E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1-3, pp. 100–118, 1985.
31. ——, "Approximate string-matching over suffix trees," in *Proc. Combinatorial Pattern Matching (CPM)*, 1993, pp. 228–242.
32. E. Viola, "The communication complexity of addition," *Combinatorica*, vol. 35, no. 6, pp. 703–747, 2015.
33. R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.
34. P. Weiner, "Linear pattern matching algorithm," in *Proc. 14th Annual IEEE Symp. on Switching and Automata Theory*, 1973, pp. 1–11.
35. W. Xia, X. Zou, H. Jiang, Y. Zhou, C. Liu, D. Feng, Y. Hua, Y. Hu, and Y. Zhang, "The design of fast content-defined chunking for data deduplication based storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2017–2031, 2020.