Near-Linear Time Insertion-Deletion Codes and (1+ε)-Approximating Edit Distance via Indexing

Bernhard Haeupler*
Carnegie Mellon University
Pittsburgh, PA, USA
haeupler@cs.cmu.edu

Aviad Rubinstein†
Stanford University
Stanford, CA, USA
aviad@cs.stanford.edu

Amirbehshad Shahrasbi*
Carnegie Mellon University
Pittsburgh, PA, USA
shahrasbi@cs.cmu.edu

ABSTRACT

We introduce *fast-decodable indexing schemes for edit distance* which can be used to speed up edit distance computations to near-linear time if one of the strings is indexed by an indexing string I. In particular, for every length n and every $\varepsilon > 0$, one can in near linear time construct a string $I \in \Sigma'^n$ with $|\Sigma'| = O_\varepsilon(1)$, such that, indexing any string $S \in \Sigma^n$, symbol-by-symbol, with I results in a string $S' \in \Sigma''^n$ where $\Sigma'' = \Sigma \times \Sigma'$ for which edit distance computations are easy, i.e., one can compute a $(1+\varepsilon)$-approximation of the edit distance between S' and any other string in $O(n\text{poly}(\log n))$ time.

Our indexing schemes can be used to improve the decoding complexity of state-of-the-art error correcting codes for insertions and deletions. In particular, they lead to near-linear time decoding algorithms for the insertion-deletion codes of [Haeupler, Shahrasbi; STOC '17] and faster decoding algorithms for list-decodable insertion-deletion codes of [Haeupler, Shahrasbi, Sudan; ICALP '18]. Interestingly, the latter codes are a crucial ingredient in the construction of fast-decodable indexing schemes.

CCS CONCEPTS

• **Mathematics of computing** → **Coding theory**; • **Theory of computation** → **Error-correcting codes**.

KEYWORDS

Coding for Insertions and Deletions, Edit Distance

ACM Reference Format:
Bernhard Haeupler, Aviad Rubinstein, and Amirbehshad Shahrasbi. 2019. Near-Linear Time Insertion-Deletion Codes and (1+ε)-Approximating Edit Distance via Indexing. In *Proceedings of the 51st Annual ACM SIGACT Symposium on the Theory of Computing (STOC '19), June 23–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3313276.3316371

4321

*Supported in part by NSF grants CCF-1618280, CCF-1814603, CCF-1527110, NSF CAREER award CCF-1750808, and a Sloan Research Fellowship.
†Supported in part by a Robert N. Noyce Family Faculty Fellowship.

STOC '19, June 23–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6705-9/19/06…$15.00
https://doi.org/10.1145/3313276.3316371

1 INTRODUCTION

Error correcting codes have revolutionized how information is stored and transmitted. The main two parameters of interest for an error correcting code are: (1) its *rate-distance trade-off*, i.e., how much redundancy is added in comparison to how many errors the code can correct and (2) its *computational efficiency*, i.e., how fast one can encode or decode. Ideal families of codes, so called near-MDS codes, over large finite alphabets achieve an (almost) perfect rate distance trade-off, i.e., attain a rate of $1-\delta-\varepsilon$ while correcting any $\delta/2$ fraction of errors (or δ fraction of erasures, insertions, or deletions) for any $\delta \in (0,1)$ and are encodable and decodable in (near) linear time.

1.1 (Near) Linear-Time Codes

The seminal works of Shannon, Hamming, and others in the late 40s and early 50s established a good understanding of the optimal rate/distance tradeoffs achievable existentially and over the next decades, near-MDS codes achieving at least polynomial time decoding and encoding procedures were put forward. Since then, lowering the computational complexity has been an important goal of coding theory. Particularly, the 90s saw a big push, spearheaded by Spielman, to achieve (near) linear coding complexities: in a breakthrough in 1994, Sipser and Spielman [35] introduced expander codes and derived linear codes with some constant distance and rate that are decodable (but not encodable) in linear time. In 1996 Spielman [36] build upon these codes to derive asymptotically good error correcting codes that are encodable and decodable in linear time. As for codes with better rate distance trade-off, Alon et al. [4] obtained near-MDS error correcting codes that were decodable from erasures in linear time. Finally, in 2004, Guruswami and Indyk [19] provided near-MDS error correcting codes for any rate than can be decoded in linear time from any combination of symbol erasures and symbol substitutions.

1.2 Codes for Insertions and Deletions

Similar questions on communication and computational efficiencies hold for synchronization codes, i.e., codes that correct from symbol insertions and symbol deletions. As a matter of fact, an analogous flow of progress can be recognized for synchronization codes. The study of synchronization codes started with the work of Levenshtein [31] in the 60s. In 1999, Schulman and Zuckerman [34] gave the first (efficient) synchronization code with constant distance and rate. Only recently, synchronization codes with stronger communication efficiency have been found. Guruswami et al. [20, 21] introduced the first synchronization codes in the asymptotically small or large noise regimes by giving efficient codes which achieve

a constant rate for noise rates going to one and codes which provide a rate going to one for an asymptotically small noise rate. Last year, Haeupler and Shahrasbi [22] were able to finally achieve efficient synchronization codes with the optimal (near-MDS) rate/distance tradeoff, for any rate and distance. These codes build on a novel tool called *synchronization strings* which are also used in [24] to design efficient list-decodable codes from insertions and deletions.

All of the codes mentioned so far have decoders with large polynomial complexity between $\Omega(n^2)$ and $O(n^{O(1/\varepsilon)})$. The only known insertion-deletion codes with subquadratic time decoders are given in [23]. Unfortunately, these codes only work for $\delta \in \left(0, \frac{1}{3}\right)$ fraction of errors while achieving a rate of $1 - 3\delta - \varepsilon$ (instead of the desired $1 - \delta - \varepsilon$).

In this work, we take the natural next step and address the problem of finding near-linear time encodable/decodable (near-MDS) codes for insertions and deletions.

1.3 Quadratic Edit Distance Computation Barrier

Many of the techniques developed for constructing efficient regular error correcting codes also apply to synchronization strings. Indeed, the synchronization string based constructions in [22–25] show that this can largely be done in a black-box manner. However, there is a serious new barrier that arises in the setting of synchronization errors if one tries to push computational complexities below n^2. This barrier becomes apparent once one notices that decoding an error correcting code is essentially doing a distance minimization where the appropriate distance in the synchronization setting is the edit distance[1]. As we discuss below, merely computing the edit distance between two strings (the input and a candidate output of the decoder) in subquadratic time is a well known hard problem. An added dimension of challenge in our setting is that we must first select the candidate decoder outputs among exponentially many codewords.

A simple algorithm for computing the edit distance of two given strings is the classic Wagner-Fischer dynamic programming algorithm that runs in quadratic time. Improving the running time of this simple algorithm has been a central open problem in computer science for decades (e.g. [16]). Yet to date, only a slightly faster algorithm ($O(n^2/\log^2 n)$) due to Masek and Paterson [32] is known. Furthermore, a sequence of complexity breakthroughs from recent years suggests that a near-quadratic running time may in fact be optimal [1–3, 8, 12] (under the Strong Exponential Time Hypothesis (SETH) or related assumptions). In order to obtain subquadratic running times, computer scientists have considered two directions: moving beyond worst-case instances, and allowing approximations.

Beyond worst case. Edit distance computation is known to be easier in several special cases. For the case where edit distance is known to be at most k, Ukkonen [37] provided an $O(nk)$ time algorithm and Landau et al. [30] improved upon that with an $O(n + k^2)$ time algorithm. For the case where the longest common subsequence (LCS) is known to be at most L, Hirschberg [27] gave an algorithm running in time $O(n \log n + Ln)$. Following a long line of works, Gawrychowski [17] currently has the fastest algorithm for the special case of strings that can be compressed as small *straight-line programs* (SLP). Andoni and Krauthgamer [5] obtain efficient approximations to edit distance for the case where the strings are perturbed a-la smoothed analysis. Goldwasser and Holden [18] obtain subquadratic algorithms when the input is augmented with auxiliary correlated strings.

Other special cases have also been considered (see also [13]), but the work closest to ours is by Hunt and Szymanski [28], who obtained a running time of $O((n+r)\log n)$ for the special case where there are r "matching pairs", i.e. pairs of identical characters (see also Section 3.2). While we directly build on their algorithm, note that there is an obstacle to applying it in our setting: for a constant size alphabet, we expect that a constant fraction of all n^2 pairs of characters will be matching, i.e. $r = \Theta(n^2)$.

Approximation algorithms. There is a long line of works on efficient approximation algorithms for edit distance in the worst case [6, 7, 9–11, 14]. First, it is important to note that even after the recent breakthrough of Chakraborty et al. [14], it is not known how to obtain approximation factors better than 3 (see also discussion in [33]). Furthermore, our running time is much faster than Chakraborty et al.'s [14] and even faster than the near-linear time approximations of [6, 7]. The best known approximation factor in time $O(n\text{polylog}(n))$ is still worse than $n^{1/3}$ [10].

In terms of techniques, our algorithm is most closely inspired by the window-compatible matching paradigm introduced by the recent quantum approximation algorithm of Boroujeni et al. [11] (a similar idea was also used by Chakraborty et al. [14]).

A new ray of hope. In this work we combine both approaches: namely we allow for (arbitrarily good) approximation, and also restrict our attention to the special case of computing the edit distance between a worst case input and a codeword from our code. The interesting question thus becomes if there is a way to build enough structure into a string (or a set of strings/codewords) that allows for fast edit distance computations. Given the importance and pervasiveness of edit distance problems we find this to be a question of interest way beyond its applicability to synchronization codes. An independent work of Kuszmaul [29] also employs the combination of the two approaches and provides a near-linear time algorithm for approximating the edit distance between a pseudo-random string and an arbitrary one within a constant factor.

2 OUR RESULTS

In this paper, we introduce a simple and generic structure that achieves this goal. In particular, we will show that there exist strings over a finite alphabet that, if one indexes any given string S with them, the edit distance of the resulting string to any other string S' can be approximated within a $1 + \varepsilon$ factor in near-linear time. This also leads to breaking the quadratic decoding time barrier for insertion-deletion codes with near-optimal communication efficiency.

We start with a formal definition of *string indexing* followed by the definition of an *indexing scheme*.

Definition 2.1 (String Indexing or Coordinate-Wise String Concatenation). Let $S \in \Sigma^n$ and $S' \in \Sigma'^n$ be two strings of length n over

[1] We define the *edit distance* between two strings S, S' as the minimum number of character insertions and deletions required to transform S to S'. Note that this is slightly different (but closely related) to the more standard definition which also allows character substitutions.

alphabets Σ and Σ'. The *coordinate-wise concatenation of S and S'* or *S indexed by S'* is a string of length n over alphabet $\Sigma \times \Sigma'$ whose ith element is (S_i, S'_i). We denote this string with $S \times S'$.

Definition 2.2 (Indexing Scheme). The pair $(I, \widetilde{\mathrm{ED}}_I)$ consisting of string $I \in \Sigma_{\mathrm{Index}}^n$ and algorithm $\widetilde{\mathrm{ED}}_I$ is an *ε-indexing scheme* if for any string $S \in \Sigma^n$ and $S' \in [\Sigma \times \Sigma_{\mathrm{Index}}]^n$, $\widetilde{\mathrm{ED}}_I(S \times I, S')$ outputs a set of up to $(1+\varepsilon)\mathrm{ED}(S \times I, S')$ symbol insertions and symbol deletions over $S \times I$ that turns it into S'. The $\mathrm{ED}(\cdot)$ notation represents the edit distance function.

The main result of this work is on the existence of indexing schemes that facilitate approximating the edit distance in near-linear time.

Theorem 2.3. *For any $\varepsilon \in (0,1)$ and integer n, there exist a string $I \in \Sigma_{\mathrm{Index}}^n$ and an algorithm $\widetilde{\mathrm{ED}}_I$ where $(I, \widetilde{\mathrm{ED}}_I)$ form an ε-indexing scheme, $|\Sigma_{\mathrm{Index}}| = \exp\left(\frac{\log(1/\varepsilon)}{\varepsilon^3}\right)$, $\widetilde{\mathrm{ED}}_I$ runs in $O_\varepsilon(n\mathrm{poly}(\log n))$ time, and I can be constructed in $O_\varepsilon(n\mathrm{poly}(\log n))$ time.*

2.1 Applications

One application of indexing schemes that we introduce in this work is in enhancing the design of insertion-deletion codes (insdel codes) from [22, 24]. The construction of codes from [22, 24] consist of indexing each codeword of some appropriately chosen error correcting code with symbols of a synchronization string which, in the decoding procedure, will be used to recover the position of received symbols. As we will recapitulate in Section 7, this procedure of recovering the positions consists of several longest common subsequence computations between the utilized synchronization string and some other version of it that is altered by a number of insertions and deletions. This fundamental step resulted in an $\Omega(n^2)$ decoding time complexity for codes in [22, 24].

Using the ε-indexing schemes in this paper, we will modify constructions of [22, 24] so that the above-mentioned longest common subsequence computations can be replaced with approximations of the longest common subsequence (using Theorem 2.3) that run in near-linear time. The following theorem, that improves the main result of [22] with respect to the decoding complexity, gives an insertion-deletion code for the entire range of distance that approaches the Singleton bound and is decodable in near-linear time.

Theorem 2.4. *For any $\varepsilon > 0$ and $\delta \in (0,1)$ there exists an encoding map $E : \Sigma^k \to \Sigma^n$ and a decoding map $D : \Sigma^* \to \Sigma^k$, such that, if $\mathrm{ED}(E(m), x) \le \delta n$ then $D(x) = m$. Further, $\frac{k}{n} > 1 - \delta - \varepsilon$, $|\Sigma| = \exp\left(\varepsilon^{-4}\log(1/\varepsilon)\right)$, and E and D are explicit and can be computed in linear and near-linear time in terms of n respectively.*

A very similar improvement is also applicable to the design of list-decodable insertion-deletion codes from [24] as they also utilize indexed synchronization strings and a similar position recovery procedure along with an appropriately chosen list-recoverable code. (See Definition 3.4) In this case, we obtain list-decodable insertion-deletion codes that match the fastest known list-recoverable codes in terms of decoding time complexity.

Theorem 2.5. *For every $0 < \delta, \varepsilon < 1$ and $\varepsilon_0, \gamma > 0$, there exists a family of list-decodable codes that can protect against δ-fraction of deletions and γ-fraction of insertions and achieves a rate of at least $1 - \delta - \varepsilon$ over an alphabet of size $O_{\varepsilon_0,\varepsilon,\gamma}(1)$. There exists a randomized decoder for these codes with list size $L_{\varepsilon_0,\varepsilon,\gamma}(n) = \exp(\exp(\exp(\log^* n)))$, $O(n^{1+\varepsilon_0})$ encoding and decoding complexities, and decoding success probability 2/3.*

Both Theorems 2.4 and 2.5 are built upon the fact that if one indexes a synchronization string with an appropriate indexing scheme, the resulting string will be a synchronization string that is decodable in near-linear time.

2.2 Other Results, Connection to List-Recovery, and Organization of the Paper

In the rest of the paper, we first provide some preliminaries and useful lemmas from previous works in Section 3. In Section 4, we introduce the construction of our indexing schemes and prove Theorem 2.3. The construction of these indexing schemes utilize insertion-deletion codes that are list-decodable from large fractions of deletions and insertions. We use list-decodable codes from [24] for that purpose, which themselves use list-recoverable codes as a core building block. Therefore, the quality of indexing schemes that we provide, namely, time complexity and alphabet size, greatly depend on utilized list-recoverable codes and can be improved following the prospective advancement of list-recoverable codes in the future.

In Section 5, we enhance the structure of the indexing scheme from Theorem 2.3 and provide Theorem 5.1 that describes a construction of indexing schemes using $(\varepsilon, \frac{1}{\varepsilon}, L)$-list-recoverable codes as a black-box. This result opens the door to potentially reduce the polylogarithmic terms in the time complexity of indexing schemes from Theorem 2.3 by future developments in the design of list-recoverable codes. For instance, finding near-linear time $(\varepsilon, \frac{1}{\varepsilon}, \mathrm{poly}(\log n))$-list recoverable codes leads to indexing schemes that run in $O(n\mathrm{poly}(\log\log n))$ time via Theorem 5.1.

As of the time of writing this paper, no such list-recoverable code is known. However, a recent work of Hemenway, Ron-Zewi, and Wootters [26] presents list-recoverable codes with $O\left(n^{1+\varepsilon_0}\right)$ time probabilistic decoders for any $\varepsilon_0 > 0$ that are appropriate for the purpose of being utilized in the construction of indexing schemes as outlined in Theorem 5.1. In Section 6, we use such codes with the indexing scheme construction method of Theorem 5.1 to provide a randomized indexing scheme with $O(n \log^{\varepsilon_0} n)$ time complexity for any chosen $\varepsilon_0 > 0$.

Then, in Section 7, we discuss the application of indexing schemes in the design of insertion-deletion codes. We start by Theorem 7.1 that enhances synchronization strings by using them along with indexing schemes and, therefore, enables us to reduce the time complexity of the position recovery subroutine of the decoders of codes from [22, 24] to near-linear time. In Section 7.2, we discuss our results for uniquely-decodable codes and prove Theorem 2.4. At the end, in Section 7.3, we address construction of list-decodable synchronization codes using indexing schemes. We start by Theorem 7.4 that gives a black-box conversion of a given list-recoverable code to a list-decodable insertion-deletion code by adding only a near-linear time overhead to the decoding complexity and, therefore, paves the path to obtaining insertion-deletion codes that are list-decodable in near-linear time upon the design

of near-linear time list-recoverable codes. We use this conversion along with list-recoverable codes of [26] to prove Theorem 2.5.

3 PRELIMINARIES AND NOTATION

In this section, we provide definitions and preliminaries that will be useful throughout the rest of the paper.

3.1 Synchronization Strings

We start by some essential definitions and lemmas regarding synchronization strings. Synchronization strings are defined as follows [22].

Definition 3.1 (ε-Synchronization Strings). String $S \in \Sigma^n$ is an ε-synchronization string if for every $1 \le i < j < k \le n+1$ we have that $\mathrm{ED}(S[i,j), S[j,k)) > (1-\varepsilon)(k-i)$.

An important property of such strings is that they cannot have pairs of long identical disjoint subsequences.

Theorem 3.2 (Theorem 6.2 of [22]). *Let S be an ε-synchronization string of length n and $1 \le i_1 < i_2 < \cdots < i_l \le n$ and $1 \le j_1 < j_2 < \cdots < j_l \le n$ be integers so that $S(i_k) = S(j_k)$ but $i_k \neq j_k$ for all $1 \le k \le l$. Then $l \le \varepsilon n$.*

We will also make use of the following construction of synchronization strings that is developed in [15, 23].

Theorem 3.3 (Theorem 1.3 from [15]). *For any $\varepsilon \in (0,1)$ and integer n, one can construct an ε-synchronization string of length n over an alphabet of size $O\left(\varepsilon^{-3}\right)$ in linear time.*

Haeupler et al. [24] suggest a construction of list-decodable insertion-deletion codes by indexing the codewords of a list-recoverable code with symbols of a synchronization string. As we will use similar techniques and ideas throughout this paper, we formally define list-recoverable codes and review the main result of [24] in the rest of this section.

Definition 3.4 (List-recoverable codes). Code C with encoding function $\mathrm{Enc} : \Sigma^{nr} \to \Sigma^n$ is called (α, l, L)-list recoverable if for any collection of n sets $S_1, S_2, \cdots, S_n \subset \Sigma$ of size l or less, there are at most L codewords for which more than αn elements appear in the list that corresponds to their position, i.e.,

$$|\{x \in C \mid |\{i \in [n] \mid x_i \in S_i\}| \ge \alpha n\}| \le L.$$

Theorem 3.5 (Theorem 1.1 from [24]). *For every $0 < \delta, \varepsilon < 1$ and $\gamma > 0$, there exist a family of list-decodable insdel codes that can protect against δ-fraction of deletions and γ-fraction of insertions and achieves a rate of $1 - \delta - \varepsilon$ or more over an alphabet of size $\left(\frac{\gamma+1}{\varepsilon^2}\right)^{O\left(\frac{\gamma+1}{\varepsilon^3}\right)} = O_{\gamma,\varepsilon}(1)$. These codes are list-decodable with lists of size $L_{\varepsilon,\gamma}(n) = \exp(\exp(\exp(\log^* n)))$, and have polynomial time encoding and decoding complexities.*

By choosing $\delta = \gamma = 1 - \epsilon$ and $\varepsilon = \epsilon/2$ in Theorem 3.5, we derive the following corollary.

Corollary 3.6. *For any $0 < \varepsilon < 1$, there exists an alphabet Σ_ε with size $\exp(\varepsilon^{-3} \log 1/\varepsilon)$ and an infinite family of insertion-deletion codes, C, that achieves a rate of $\frac{\varepsilon}{2}$ and is L-list-decodable from any $(1-\varepsilon)n$ deletions and $(1-\varepsilon)n$ insertions in polynomial time where $L = \exp(\exp(\exp(\log^* n)))$.*

3.2 Non-crossing Matchings

The last element that we utilize as a preliminary in this paper is an algorithm provided in a work of Hunt and Szymanski [28] to compute the maximum *non-crossing matching* in a bipartite graph. Let G be a bipartite graph with an ordering for vertices in each part. A non-crossing matching in G is a matching in which edges do not intersect.

Definition 3.7 (Non-Crossing Matching). Let G be a bipartite graph with ordered vertices $u_1, u_2, \cdots, u_m$ and $v_1, v_2, \cdots, v_n$ in each part. A non-crossing matching is a subset of edges of G like

$$\left\{(u_{i_1}, v_{j_1}), (u_{i_2}, v_{j_2}), \cdots, (u_{i_l}, v_{j_l})\right\}$$

where $i_1 < i_2 < \cdots < i_l$ and $j_1 < j_2 < \cdots < j_l$.

In this paper, we use an algorithm by Hunt and Szymanski [28] that essentially computes the largest non-crossing matching in a given bipartite graph.

Theorem 3.8 (Theorem 2 of Hunt and Szymanski [28]). *Let G be a bipartite graph with n ordered vertices in each part and r edges. There is an algorithm that computes the largest non-crossing matching of G in $O((n+r)\log\log n)$.*

4 NEAR-LINEAR EDIT DISTANCE COMPUTATIONS VIA INDEXING

We start by a description of the string that will be used in our indexing scheme. Let C be an insertion-deletion code over alphabet Σ_C, with block length N, and rate r that is L-list decodable from any $N(1-\varepsilon)$ deletions and $N(1-\varepsilon)$ insertions in $T_{\mathrm{Dec}_C}(N)$ for some sufficiently small $\varepsilon > 0$. We construct the indexing sequence I by simply concatenating the codewords of C. The construction of such indexing sequence resembles long-distance synchronization strings from [23].

Throughout this section, we consider string S of length $N \cdot |\Sigma_C|^{Nr}$ that consists of coordinate-wise concatenation of a content string m and the indexing string I. In other words, $S_i = (m_i, I_i)$. We will provide algorithms that approximate the edit distance of S to a given string S'.

Consider the longest common subsequence between S and S'. One can represent such common subsequence by a matching $\mathcal{M}_{LCS}$ with non-crossing edges in a bipartite graph with two parts of size $|S|$ and $|S'|$ where each vertex corresponds to a symbol in S or S' and each edge corresponds to a pair of identical symbols in the longest common subsequence.

Note that one can turn S into S' by simply deleting any symbol that corresponds to an unmatched vertex in S and then inserting symbols that correspond to the unmatched vertices in S'. Therefore, the edit distance between S and S' is equal to the number of non-connected vertices in that graph. To provide a $(1+\varepsilon)$ edit distance approximation as described in Theorem 2.3, one only needs to compute a common subsequence, or equivalently, a non-crossing matching between S and S' in which the number of unmatched vertices does not exceed a $1+\varepsilon$ multiplicative factor of $\mathcal{M}_{LCS}$'s.

We start by an informal intuitive justification of the algorithm. The algorithm starts by splitting the string S' into blocks of length N in the same spirit as S. We denote ith such block by $S'(i)$ and the ith block of S by $S(i)$. Note that the blocks of S are codewords

of an insertion-deletion code with high distance indexed by m ($S(i) = C(i) \times m[N(i-1), Ni-1]$). Therefore, one might expect that any block of S that is not significantly altered by insertions and deletions, (1) appears in a set of consecutive blocks in S' and (2) has a small edit distance to at least one of those blocks.

Following this intuition, our proposed algorithm works thusly: For any block of S' like $S'(i)$, the algorithm uses the list decoder of C to find all (up to L) blocks of S that can be turned into $S'(i)$ by $N(1-\varepsilon)$ deletions and $N(1-\varepsilon)$ insertions ignoring the content portion on S'. In other words, let $S'(i) = C'_i \times m'[N(i-1), Ni-1]$. We denote the set of such blocks by $\mathrm{Dec}_C(C'_i)$. Then, the algorithm constructs a bipartite graph G with $|S|$ and $|S'|$ vertices on each side (representing symbols of S and S') as follows: a symbol in $S'(i)$ is connected to all identical symbols in the blocks that appear in $\mathrm{Dec}_C(C'_i)$ or any block that is in their $w = O(1/\varepsilon)$ neighborhood, i.e., is up to $O(1/\varepsilon)$ blocks away from at least one of the members of $\mathrm{Dec}_C(C'_i)$.

Note that any non-crossing matching in G corresponds to some common subsequence between S and S' because G's edges only connect identical symbols. In the next step, the algorithm finds the largest non-crossing matching in G, $\mathcal{M}_{ALG}$, and outputs the corresponding set of insertions and deletions as the output. We will use the algorithm proposed by Hunt and Szymanski [28] (see Theorem 3.8) to find the largest non-crossing matching. A formal description of the algorithm is available in Algorithm 1.

Algorithm 1 $(1+11\varepsilon)$-Approximation for Edit Distance

```
1:  procedure ED-Approx(S, S', N, Dec_C(·))
2:      Make empty bipartite graph G with parts of size (|S|, |S'|)
3:      w = 1/ε
4:      for each S'(i) = C'_i × m'[N(i−1), Ni−1] do
5:          List ← Dec_C(C'_i)
6:          for each j ∈ List do
7:              for k ∈ [j − w, j + w] do
8:                  Connect pairs of vertices in G that correspond
    to identical symbols in S(k) and S'(i).
9:      M_ALG ← Largest non-crossing matching in G
10:     return M_ALG
```

4.1 Analysis

We now proceed to the analysis of approximation guarantee and time complexity of Algorithm 1.

Theorem 4.1. *For $n = \max(|S|, |S'|)$, the running time of Algorithm 1 is $O\left(\frac{n}{N} \cdot T_{\mathrm{Dec}_C}(N) + \frac{NL}{\varepsilon} \cdot n \log\log n\right)$.*

Proof. The algorithm starts by using the decoder for any block in S' which takes a total of $\frac{n}{N} \cdot T_{\mathrm{Dec}_C}(N)$ time. Further, construction of G will take $O(nLN/\varepsilon)$. G has no more than $n \cdot NL \cdot w = O(nNL/\varepsilon)$ edges. Thus, by using Hunt and Szymanski's [28] algorithm (Theorem 3.8), the maximum non-crossing matching in G can be computed in $O\left(\frac{NL}{\varepsilon} \cdot n \log\log n\right)$. □

Before providing the analysis for the approximation ratio of Algorithm 1, we define the following useful notions.

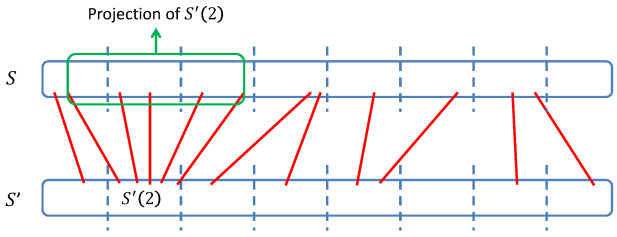


Figure 1: An example of a matching between S and S' depicting the projection of $S'(2)$. This matching is 3-window-limited.

Definition 4.2 (Projection). Let $\mathcal{M}$ be a non-crossing matching between S and S'. The *projection of $S'(i)$ under $\mathcal{M}$* is defined to be the substring of S between the leftmost and the rightmost element of S that are connected to $S'(i)$ in $\mathcal{M}$. (see Fig. 1 for an example)

Definition 4.3 (Window Limited). A non-crossing matching between S and S' is called *w-window-limited* if the projection of any block of S' fits in w consecutive blocks of S.

The definition of window-limited matchings is inspired by the window-compatibility notion from [11].

Theorem 4.4. *For $0 < \varepsilon < \frac{1}{21}$, Algorithm 1 computes a set of up to $(1+11\varepsilon) \cdot \mathrm{ED}(S, S')$ insertions and deletions that turn S into S'.*

Proof. Let ED_{ALG} denote the edit distance solution obtained by the matching suggested by Algorithm 1. We will prove that $\mathrm{ED}_{ALG} \le (1+11\varepsilon) \cdot \mathrm{ED}(S, S')$ in the following two steps:

(1) Let $\mathcal{M}_W$ be the largest $w = \left(\frac{1}{\varepsilon} + 1\right)$-window-limited matching between S and S' and ED_W be its count of unmatched vertices. In the first step, we show the following.

$$\mathrm{ED}_W \le (1+3\varepsilon)\mathrm{ED}(S, S') \tag{1}$$

To prove this, consider $\mathcal{M}_{LCS}$, the matching that corresponds to the longest common subsequence. Then, we modify this matching by deleting all the edges connected to any block $S'(i)$ that violates the w-window-limited requirement. In other words, if the projection of $S'(i)$ spans over at least $w+1$ blocks in S, we remove all the edges with one endpoint in $S'(i)$. Note that removing the edges connected to $S'(i)$ might increase the number of unmatched vertices in the matching by $2N$. However, as projection of $S'(i)$ spans over at least $w+1$ blocks in S, one can assign all the originally unmatched vertices in that projection, which are at least $(w-1) \cdot N - N \ge (w-2)N$, to the newly introduced unmatched edges as an "approximation budget". Note that this assignment is mutually exclusive since projections of two distinct blocks of S' are disjoint. Therefore, the above-mentioned removal procedure increases the number of unmatched vertices by a multiplicative factor no larger than $\frac{(w-2)N+2N}{(w-2)N} = \frac{w}{w-2} = \frac{1+\varepsilon}{1-\varepsilon} \le 1+3\varepsilon$ for $\varepsilon \le \frac{1}{3}$.

Note that the matching obtained by the above-mentioned removal procedure is a w-window-limited matching and, therefore, has at least ED_W unmatched vertices by the definition of $\mathcal{M}_W$. Hence, Eq. (1) is proved.

(2) In the second step, we show that

$$\mathrm{ED}_{ALG} \le (1+7\varepsilon)\mathrm{ED}_W. \tag{2}$$

Similar to Step 1, consider the largest w-window-limited matching $\mathcal{M}_W$ and then modify it by removing all the edges connected to any block $S'(i)$ that has less than εN edges to any block in S. Again, we prove an approximation ratio by exclusively assigning some of the unmatched vertices in $\mathcal{M}_W$ to each $S'(i)$ that we choose to remove its edges.

Consider some $S'(i)$ that has less than εN edges to any block in S. We assign all unmatched vertices in $S'(i)$ and all unmatched vertices in the projection of $S'(i)$ as the approximation budget for eliminated edges. Let B be the number of blocks in S that are contained or intersect with projection of $S'(i)$. As $S'(i)$ has less than εN edges to any block in S, the total number of removed edges is less than $NB\varepsilon$. This gives that there are at least $N - NB\varepsilon$ unmatched vertices within S' and $\max\{B-2,0\}\cdot N(1-\varepsilon)$ unmatched vertices in its projection that are assigned to $2NB\varepsilon$ new unmatched edges appearing as a result of removing $S'(i)$'s edges. Therefore, this process does not increase the number of unmatched vertices by a multiplicative factor more than $1+\frac{2NB\varepsilon}{(N-NB\varepsilon)+\max\{B-2,0\}\cdot N(1-\varepsilon)}$. If $B=1$ or 2, the above approximation ratio can be bounded above by $1+\frac{2NB\varepsilon}{(N-NB\varepsilon)} \le 1+\frac{4\varepsilon}{1-2\varepsilon} \le 1+5\varepsilon$ for $\varepsilon \le \frac{1}{10}$. Unless, $B \ge 3$, therefore the approximation ratio is less than $1+\frac{2NB\varepsilon}{(B-2)N(1-\varepsilon)} \le 1+\frac{6\varepsilon}{1-\varepsilon} \le 1+7\varepsilon$ for $\varepsilon \le \frac{1}{7}$. Therefore, the edge removal process in Step 2 does not increase the number of unmatched vertices by a factor larger than $1+7\varepsilon$.

Note that the matching obtained after the above-mentioned procedure is a w-window limited one in which any block of S' that contains at least one edge, has more than $N\varepsilon$ edges to some block in S within its projection. Therefore, this matching is a subgraph of G. Since $\mathcal{M}_{ALG}$ is defined to be the largest non-crossing matching in G, the number of unmatched vertices in $\mathcal{M}_{ALG}$, ED_{ALG} is not larger than the ones in the matching we obtained in Step 2. Hence, proof of Eq. (2) is complete.

Combining Eqs. (1) and (2) implies the following approximation ratio.

$$\mathrm{ED}_{ALG} \le (1+3\varepsilon)(1+7\varepsilon)\mathrm{ED}(S,S') \le (1+11\varepsilon)\mathrm{ED}(S,S') \tag{3}$$

The last inequality holds for $\varepsilon \le \frac{1}{21}$. □

4.2 Proof of Theorem 2.3

Proof. To prove this theorem, take $\varepsilon' = \frac{\varepsilon}{11}$. Further, take C as an insertion-deletion code from Theorem 3.5 with block length $N = c_0 \cdot \frac{\log n \cdot \varepsilon'^3}{(1-2\varepsilon')\log(1/\varepsilon')}$ and parameters $\delta_C = \gamma_C = 1-\varepsilon'$, $\varepsilon_C = \varepsilon'$. (constant c_0 will be determined later)

According to Theorem 3.5, C is $O_\varepsilon(\exp(\exp(\exp(\log^* n))))$-list decodable from $(1-\varepsilon')N$ insertions and $(1-\varepsilon')N$ deletions, is over an alphabet of size $q_C = \varepsilon'^{-O(1/\varepsilon'^3)} = \exp\left(\frac{\log(1/\varepsilon')}{\varepsilon'^3}\right)$, and has rate $r_C = 1-2\varepsilon'$.

Construct string I according to the structure described in the beginning of Section 4 using C as the required list-decodable insertion-deletion code. Note that $|I| = N \cdot q_C^{r_C N} = N \cdot \exp(c_0 \cdot O(\log n))$. Choosing an appropriate constant c_0 that cancels out the constants hidden in O-notation that originate from hidden constants in the alphabet size will lead to $|I| = Nn = O(n \log n)$. Truncate the extra elements to have string I of length n. As C is efficiently encodable, string I can be constructed in near-linear time.

Further, define algorithm $\widetilde{\mathrm{ED}}_I$ as follows. $\widetilde{\mathrm{ED}}_I$ takes $S \times I$ and S' and runs an instance of Algorithm 1 with $S \times I$, S', N, and the decoder of C as its input. Theorem 4.4 guarantees that $\widetilde{\mathrm{ED}}_I(S\times I, S')$ generates a set of at most $(1+11\varepsilon')\mathrm{ED}(S\times I, S') = (1+\varepsilon)\mathrm{ED}(S\times I, S')$ insertions and deletions over $S \times I$ that converts it to S'. Finally, Theorem 4.1 guarantees that $\widetilde{\mathrm{ED}}_I$ runs in

$$\begin{aligned} & O\left(\frac{n}{N}\cdot T_{\mathrm{Dec}_C}(N) + \frac{NL}{\varepsilon}\cdot n\log\log n\right) \\ = \; & O_\varepsilon\left(\frac{nT_{\mathrm{Dec}_C}(\log n)}{\log n} + n\log n\log\log n\exp(\exp(\exp(\log^* n)))\right) \\ = \; & O_\varepsilon(n\mathrm{poly}(\log n)) \end{aligned}$$

time. □

5 ENHANCED INDEXING SCHEME

In Section 4, we provided an indexing scheme, using which, one can essentially approximate the edit distance by a $(1+\varepsilon)$ multiplicative factor for any $\varepsilon > 0$. Note that if code C that was used in that construction has some constant rate $r = O_\varepsilon(1)$, then $|S| = N \cdot |\Sigma_C|^{Nr}$ and, therefore, $N = \Theta_\varepsilon(\log n/r)$. This makes the running time of Algorithm 1 from Theorem 4.1 $O(nr/\log n \cdot T_{\mathrm{Dec}_C}(\log n) + \log n \cdot L/\varepsilon \cdot n\log\log n)$. As described in the proof of Theorem 2.3, using the efficient list-decodable codes from Corollary 3.6, one can obtain edit distance computations in $O_\varepsilon(n\cdot\mathrm{poly}(\log n) + n\log n \cdot \log\log n \cdot \exp(\exp(\exp(\log^* n)))) = O_\varepsilon(n\cdot\mathrm{poly}(\log n))$.

In this section, we try to enhance this running time by reducing the poly-logarithmic terms. To this end, we break down the factors in our construction and edit distance computation that contribute to the poly-logarithmic terms in the decoding time complexity.

(1) **Edges in graph** G: The number of edges in graph G can be as high as $\Theta(nNL/\varepsilon) = \Theta(n\log n \cdot \mathrm{poly}(\log\log n))$ which, as discussed above, leads to an additive $n\log n \cdot \log\log n \cdot \exp(\exp(\exp(\log^* n)))$ component. In Section 5.1, we will show that this component can be reduced to $O(n\cdot\mathrm{poly}(\log\log n))$ by having two layers of indices via indexing each codeword of C with an indexing scheme as described in Section 4 (constructed based on some code of block length $O(\log\log n)$).

(2) **Decoding complexity of** C **from Corollary 3.6** ($T_{\mathrm{Dec}_C}(\cdot)$): As described in Section 3.1, list-decodable insdel codes from Theorem 3.5 are obtained by indexing codewords of a list-recoverable code with a synchronization string and their decoding procedure consist of (1) calculating a constant number of longest common subsequence computations, and (2) running the decoder of the list-recoverable code.

Part (1) consumes quadratic time in terms of N. However, using the indexing schemes for approximating edit distance from Theorem 2.3, we will show in Theorem 7.4 that one can reduce the running time of part (1) to $O\left(\frac{n}{\log n} \cdot \log n \cdot \text{poly}(\log\log n)\right) = O\left(n \cdot \text{poly}(\log\log n)\right)$.

Applying the above-metioned enhancements to the structure of our indexing scheme will result in the black-box construction of indexing schemes using list-recoverable codes as formalized in the following theorem.

Theorem 5.1. *For any $\varepsilon \in (0,1)$, given a family of codes over alphabet Σ that are $\left(\frac{\varepsilon}{46}, \frac{276}{\varepsilon}, L(\cdot)\right)$-list recoverable in $T_{\text{Dec}}(\cdot)$ time and achieve a rate of $r = O_\varepsilon(1)$, one can construct an ε-indexing scheme $(I, \widetilde{\text{ED}}_I)$ with any positive length n over an alphabet of size $|\Sigma|^2 \times \exp\left(\frac{\log(1/\varepsilon)}{\varepsilon^3}\right)$ where $\widetilde{\text{ED}}_I$ has $O_\varepsilon\left(n \cdot \left[\frac{T_{\text{Dec}}(\log n)}{\log n} + \frac{T_{\text{Dec}}(\log\log n)}{\log\log n} + \log^2\log n \cdot L(\log n) L(\log\log n) + \text{poly}(\log\log n)\right]\right)$ running time complexity. Further, if the given family of codes are efficiently encodable, I can be constructed in near-linear time.*

These enhancements do not eventually yield an indexing scheme that works in $O(n \cdot \text{poly}(\log\log n))$ as the bottleneck of the indexing scheme's time complexity is the decoding time of the utilized list-recoverable code.

As of the time of writing this paper, no deterministic list recoverable code with our ideal properties and a decoding time complexity faster than an unspecified large polynomial is found. However, because of the enhancements discussed in this section, improvements in decoding time complexity of list-recoverable codes can lead to ε-indexing schemes that run in $O(n \cdot \text{poly}(\log\log n))$ time. Particularly, having a linear-time $(\varepsilon, 1/\varepsilon, L(n) = \text{poly}(\log n))$-list recoverable code would suffice.

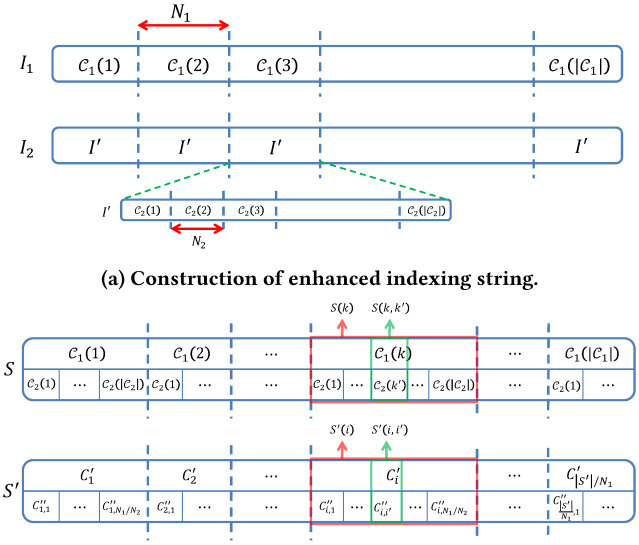
5.1 Two Layer Indexing

Our enhanced indexing sequence I consists of the coordinate-wise concatenation of two string I_1 and I_2 where I_1 is the ordinary indexing sequence as described in Section 4, i.e, the codewords of a code C_1 with block length N_1, and I_2 is repetitions of an ordinary indexing sequence I' of length N_1 constructed using some code C_2. (See Fig. 2a)

In other words, let C_2 be a code of block length N_2 and rate r_2 over alphabet Σ_{C_2} that is L_2-list decodable from $N_2(1-\varepsilon)$ insertions and $N_2(1-\varepsilon)$ deletions. Writing the codewords of C_2 back to back would give the string I' of length $|I'| = N_2 \cdot |\Sigma_{C_2}|^{N_2 r_2}$. Then, let code C_1 be a code of block length $N_1 = |I'|$ and rate r_1 over alphabet Σ_{C_1} that is L_1-list decodable from $N_1(1-\varepsilon)$ insertions and $N_1(1-\varepsilon)$ deletions. We form string I_1 by writing the codewords of C_1 one after another and string I_2 by repeating I' for $|C_1|$ times. Finally, $I = (I_1, I_2)$.

We provide a decoding algorithm for indexing sequence I that is very similar to Algorithm 1 with an extra step in the construction of bipartite graph G that reduces the number of edges at the cost of a weaker yet still constant approximation guarantee.

In Line 8 of Algorithm 1, instead of adding an edge between any two pair of identical symbols in $S(k)$ and $S'(i)$ (that can be as many as $\log^2 n$), the algorithm runs another level of list-decoding and window-limiting based on the copy of I' that is a component of $S(k)$. In other word, the algorithm uses the decoder of C_2 for any sub-block of length N_2 in $S'(i)$, like $S'(i,i')$, to find up to L_2 sub-blocks of length N_2 in $S(k)$, like $S(k,k')$, and adds an edge between any two identical symbols between $S'(i,i')$ and $S(k,k')$. We denote the portion of $S'(i,i')$ that corresponds to C_2 codewords by $C''_{i,i'}$. (See Fig. 2b) A formal description is available in Algorithm 2.



(a) Construction of enhanced indexing string.

(b) Decoding for enhanced construction.

Figure 2

Algorithm 2 (1 + 23ε)-Approximation for Edit Distance

procedure Enhanced-ED-Approx($S, S', \{N_i, \text{Dec}_{C_i}(\cdot)\}_{i=1}^2$)
  Make empty bipartite graph G with parts of size $|S|$ and $|S'|$
  $w = \frac{1}{\varepsilon}$
  for each $S'(i) = C'_i \times \left[C''_{i,1}, C''_{i,2}, \cdots, C''_{i,N_1/N_2}\right] \times m'[N_1(i-1), N_1 i - 1]$ **do**
    $List_1 \leftarrow \text{Dec}_{C_1}(C'_i)$
    for each $j \in List_1$ **do**
      for $k \in [j-w, j+w]$ **do**
        for $i' \in [1, N_1/N_2]$ **do**
          $List_2 \leftarrow \text{Dec}_{C_2}(C''_{i,i'})$
          for each $j' \in List_2$ **do**
            for $k' \in [j'-w, j'+w]$ **do**
              Connect any pair of vertices in G that correspond to identical symbols in $S(k,k')$ and $S'(i,i')$.
  $\mathcal{M}_{ALG} \leftarrow$ Largest non-crossing matching in G
  return $\mathcal{M}_{ALG}$

Theorem 5.2. *For $n = \max(|S|, |S'|)$, Algorithm 2 runs in $O\left(\frac{n}{N_1} \cdot T_{\text{Dec}_{C_1}}(N_1) + \frac{n}{N_2} \cdot T_{\text{Dec}_{C_2}}(N_2) + \frac{N_2 L_1 L_2}{\varepsilon^2} \cdot n \log\log n\right)$ time.*

Proof. The algorithm uses the decoder of C_1, $\frac{n}{N_1}$ times and the decoder of C_2, $\frac{n}{N_2}$ times. G can have up to $\frac{n}{N} \cdot \frac{L_1}{\varepsilon} \cdot \frac{N_1}{N_2} \cdot \frac{L_2}{\varepsilon} \cdot N_2^2 = \frac{N_2 L_1 L_2}{\varepsilon^2} \cdot n$ edges. Therefore, the use of Hunt and Szymanski's [28] algorithm (Theorem 3.8) will take $O\left(N_2 L_1 L_2/\varepsilon^2 \cdot n \log\log n\right)$ time. Therefore, the time complexity is as claimed. □

Theorem 5.3. *For $0 < \varepsilon < \frac{1}{121}$, Algorithm 2 computes a set of up to $(1 + 23\varepsilon) \cdot \mathrm{ED}(S, S')$ insertions and deletions that turn S into S'.*

Proof. In the proof of Theorem 4.4, we proved that for the graph G in Algorithm 1, $\mathrm{ED}_{ALG} \le (1 + 11\varepsilon)\mathrm{ED}(S, S')$. In other words, the number of unmatched vertices in the largest non-crossing matching in that graph is at most $(1+11\varepsilon)$ times the number of unmatched vertices in the bipartite graph that corresponds to the longest common subsequence between S and S'.

As graph G in Algorithm 2 is the same as the one in Algorithm 1 with some extra edges removed, we only need to show that removing the extra edges does not increase the number of non-matched vertices in the largest non-crossing matching by more than a $(1 + O(\varepsilon))$ multiplicative factor. This can be directly concluded from Theorem 4.4 since the extra removed edges are eliminated by doing the same procedure between pairs of codewords of C_2 that is done between the strings in the statement of Theorem 4.4. In fact, using similar budget-based arguments as in Eqs. (1) and (2), the extra edge removal step will only increase the edit distance by a $(1 + 11\varepsilon)$ factor. This leads to an upper bound of $(1 + 11\varepsilon)(1 + 11\varepsilon)\mathrm{ED}(S, S') \le (1 + 23\varepsilon)\mathrm{ED}(S, S')$ on the approximation ratio of Algorithm 2 for $\varepsilon < \frac{1}{121}$. □

5.2 Proof of Theorem 5.1

Proof. Let $\varepsilon' = \varepsilon/46$. Thus, the given family of codes is $(\varepsilon', 6/\varepsilon', L(\cdot))$-list recoverable.

Take the code C_1 as a code with block length N_1 from the given family of codes where N_1 is large enough so that $N_1 \cdot |\Sigma|^{r/2 \cdot N_1} \ge n$. Similarly, take C_2 with block length N_2 so that $N_2 \cdot |\Sigma|^{r/2 \cdot N_2} \ge N_1$. For a large enough n, rates of C_1 and C_2 are at least $r/2$. We reduce the rates of C_1 and C_2 to $r/2$ by arbitrarily removing codewords from them.

We now use Theorem 7.4 with parameters $\varepsilon_{\mathrm{conv}} = \varepsilon'$ and $\gamma_{\mathrm{conv}} = 1 - 2\varepsilon'$ to convert list-recoverable codes C_1 and C_2 to list-decodable insertion-deletion codes $\tilde{C}_1$ and $\tilde{C}_2$ by indexing their codewords with appropriately chosen indexing sequences from Theorem 2.3 and synchronization strings. Note that we can do this conversion using Theorem 7.4 since $\gamma_{\mathrm{conv}} = 1-2\varepsilon' \le \frac{l_{C_i} \cdot \varepsilon_{\mathrm{conv}}}{3} - 1 = \frac{6/\varepsilon' \cdot \varepsilon'}{3} - 1 = 1$. Also, $\tilde{C}_i$ can $L(N_i)$-list decode from any $\gamma_{\mathrm{conv}} = 1 - 2\varepsilon'$ fraction of insertions and any $1 - \alpha_{C_i} - \varepsilon_{\mathrm{conv}} = 1 - \frac{\varepsilon}{46} - \varepsilon' = 1 - 2\varepsilon'$ fraction of deletions in $T_{\mathrm{Dec}}(N_i) + O\left(N_i \mathrm{poly}(\log N_i)\right)$.

Also, it is known how to construct ε_s-synchronization strings and ε_I-indexing schemes needed in Theorem 7.4. ε_s-synchronization strings can be constructed in linear time in terms of their length over an alphabet of size $\varepsilon_s^{-O(1)}$ and ε_I-indexing sequences from Theorem 2.3 can be constructed in near-linear time over an alphabet of size $\exp\left(\log(1/\varepsilon_I)/\varepsilon_I^3\right)$. Therefore, the alphabets of $\tilde{C}_1$ and $\tilde{C}_2$ will be of size $|\Sigma| \times \exp\left(\log(1/\varepsilon')/\varepsilon'^3\right)$.

We now use codes $\tilde{C}_1$ and $\tilde{C}_2$ in the structure described in the beginning of Section 5.1 to obtain an indexing sequence I of length n. Since the conversion of each codeword of C_i to $\tilde{C}_i$ consumes near-linear time in terms of N_i, if the codes C_i are efficiently encodable, string I can be constructed in near-linear time. Also, the above-mentioned discussion on alphabet sizes of $\tilde{C}_i$ entails that I will be a string over an alphabet of size $|\Sigma|^2 \times \exp\left(\log(1/\varepsilon')/\varepsilon'^3\right)$.

We now have to provide an algorithm that produces a $(1 + \varepsilon)$-approximation for the edit distance using I. In the same spirit as the algorithm provided in the proof of Theorem 5.1, we define algorithm $\widetilde{\mathrm{ED}}_I$ as an algorithm that takes $S \times I$ and S' and runs an instance of Algorithm 2 with $S \times I$, S', N_1, N_2, and decoders of $\tilde{C}_i$ as its input.

As codes $\tilde{C}_i$ list decode from $1 - 2\varepsilon'$ fraction of insertions and deletions, Theorem 5.3 guarantees that $\widetilde{\mathrm{ED}}_I$ generates a set of at most $(1 + 23 \cdot 2(\varepsilon'))\mathrm{ED}(S \times I, S') = (1 + \varepsilon)\mathrm{ED}(S \times I, S')$ insertions and deletions over $S \times I$ that converts it to S'.

Finally, since $N_1 = O(\log n)$, $N_2 = O(\log\log n)$ and $\tilde{C}_i$ list decode in $T_{\mathrm{Dec}}(N_i) + O\left(N_i \mathrm{poly}(\log N_i)\right)$ time, Theorem 5.2 guarantees that $\widetilde{\mathrm{ED}}_I$ runs in

$$O\left(\frac{n}{N_1} \cdot T_{\mathrm{Dec}_{\tilde{C}_1}}(N_1) + \frac{n}{N_2} \cdot T_{\mathrm{Dec}_{\tilde{C}_2}}(N_2) + \frac{N_2 L_1 L_2}{\varepsilon^2} \cdot n \log\log n\right)$$

time. This gives the running time in the theorem statement. □

6 RANDOMIZED INDEXING

In this section, we will prove the following theorem by taking similar steps as in the proof of Theorem 5.1 to construct an indexing scheme according to the structure introduced in Section 5.1.

Theorem 6.1. *For any $\varepsilon_0 > 0$, $\varepsilon_1, \varepsilon_2 \in (0, 1)$, and integer n, there exists a randomized indexing scheme $(I, \widetilde{\mathrm{ED}}_I)$ of length n where $\widetilde{\mathrm{ED}}_I(S \times I, S')$ runs in $O(n \log^{\varepsilon_0} n)$ time and proposes a set of insertions and deletions that turns $S \times I$ into S' and contains up to $(1 + \varepsilon_1)\mathrm{ED}(S \times I, S') + \varepsilon_2|S'|$ operations with probability $1 - \frac{1}{n^{O(1)}}$.*

Note that, as opposed to the rest of the results in this paper, Theorem 6.1 provides an approximation guarantee with both multiplicative and additive components.

To construct such an indexing scheme using the structure introduced in Section 5.1, we will use a list-decodable insertion-deletion code of block length $O(\log\log n)$ from Corollary 3.6 and use Theorem 7.4 to obtain a list-decodable insertion-deletion code of block length $O(\log n)$ from the following list recoverable codes of [26].

Theorem 6.2 (Corollary of Theorem 7.1 of Hemenway et al. [26]). *For any $\rho \in [0, 1]$, $\varepsilon > 0$, and positive integer l, there exist constants q_0 and c_0 so that, for any $c < c_0$ and infinitely many integers $q \ge q_0$, there exists an infinite family of codes achieving the rate ρ over an alphabet Σ of size $|\Sigma| = q$ that is encodable in n^{1+c} time and probabilistically $(\rho + \varepsilon, l, L(n))$-list recoverable in n^{1+c} time with success probability 2/3 and $L(n) = O_{\varepsilon,\rho}(\exp(\exp(\exp(\log^* n))))$ where n denotes the block length.*

Before providing the proof of Theorem 6.1, we mention a couple of necessary lemmas.

Lemma 6.3. *Let $(\alpha, l, L(n))$-list-recoverable code C have a probabilistic decoder that runs in $T_{\mathrm{Dec}}(n)$ and works with probability p. Then,*

for any integer k, C can be $(\alpha, l, k \cdot L(n))$-list-recovered in $kT_{\mathrm{Dec}}(n)$ time with $1-(1-p)^k$ success probability.

Proof. Use a decoding procedure for C that repeats the given decoder k times and outputs the union of the lists produced by them. The final list size will be at most $kL(n)$ long, the running time will be $kT_{\mathrm{Dec}}(n)$, and the failure probability, i.e., the probability of the output list not containing the correct codeword is at most $(1-p)^k$. □

Another required ingredient to the proof of Theorem 6.1 is to show how a probabilistic decoder affect the approximation guarantee of Algorithm 2. To this end, we provide the following lemma as an analogy of Theorem 5.3 when the decoder of code C_1 is not deterministic.

Lemma 6.4. *Let the decoder of code C_1 ($\mathrm{Dec}_{C_1}(\cdot)$) be a randomized algorithm that L_1-list decodes the code C_1 with probability $1-p$. Then, with probability $1-e^{-\frac{2|S'|p}{3N_1}}$, Algorithm 2 will generate a set of up to $(1+23\varepsilon)\mathrm{ED}(S,S')+2p|S'|$ insertions and deletions that turn S into S'.*

Proof. If $\mathrm{Dec}_{C_1}(\cdot)$ worked with probability 1, the outcome of $\mathcal{A}$ would contain up to $(1+23\varepsilon_1)$ insertions and deletions. Each time that Dec_{C_1} fails to correctly list-decode a block of length N_1 (C_i'), up to N_1 edges from $\mathcal{M}_{ALG}$ might be lost and, consequently, there can be up to $2N_1$ units of increase in the number of insertions and deletions generated by $\mathcal{A}$.

There are a total of $n=|S'|/N_1$ list decodings and each might fail with probability p. Using the Chernoff bound,

$$\Pr(\text{more than } 2np \text{ failures}) \le e^{-2np/3} = e^{-\frac{2|S'|p}{3N_1}}.$$

Thus, with probability $1-e^{-\frac{2|S'|p}{3N_1}}$, the output of $\mathcal{A}$ contains $(1+23\varepsilon)\mathrm{ED}(S,S')+2npN_1=(1+23\varepsilon)\mathrm{ED}(S,S')+2p|S'|$ or less insertions and deletions. □

We are now adequately equipped to prove Theorem 6.1.

6.1 Proof of Theorem 6.1

Proof. Our construction closely follows the steps taken in the proof of Theorem 5.1. Let $\varepsilon'=\varepsilon_1/46$. Take C_1 from the Theorem 6.2 with parameters $\varepsilon_{C_1}=\varepsilon'$, $\rho_{C_1}=2\varepsilon'$, $l_{C_1}=6/\varepsilon'$, $c_{C_1}=\varepsilon_0$, and block length N_1 where N_1 is large enough so that $N_1 \cdot q_{C_1}^{\rho_{C_1}/2\cdot N_1} \ge n$ where q_{C_1} is the size of the alphabet of the family codes.

According to Theorem 6.2, C_1 is probabilistically $(\varepsilon', \varepsilon'/6, L(N_1))$-list recoverable in $O_{\varepsilon_1}(N_1^{1+\varepsilon_0})$ time where $L(N_1)=\exp(\exp(\exp(\log^* N_1)))$ and success probability is 2/3. We use Lemma 6.3 with repetition number parameter $k=\log_3\frac{2}{\varepsilon_2}$ to obtain a $(\varepsilon', \varepsilon'/6, O(\log(1/\varepsilon_2)\cdot L(N_1)))$-list recovery algorithm for C_1 that succeeds with probability $1-(\frac{1}{3})^k = 1-\frac{\varepsilon_2}{2}$ and runs in $O_{\varepsilon_1}\left(N_1^{1+\varepsilon_0}/\varepsilon_2\right)$ time.

We now use Theorem 7.4 with parameters $\varepsilon_{\mathrm{conv}}=\varepsilon'$ and $\gamma_{\mathrm{conv}}=1-2\varepsilon'$ to convert list-recoverable code C_1 to a list-decodable insertion-deletion code $\tilde{C}_1$ by indexing its codewords with an appropriately chosen indexing sequence from Theorem 2.3 and a synchronization string. Note that we can do this conversion using Theorem 7.4 since $\gamma_{\mathrm{conv}}=1-2\varepsilon' \le \frac{l_{C_1}\cdot\varepsilon_{\mathrm{conv}}}{3}-1=\frac{6/\varepsilon'\cdot\varepsilon'}{3}-1=1$. Also, $\tilde{C}_1$ can $O\left(\log\frac{1}{\varepsilon_2}L(N_1)\right)$-list decode from any $\gamma_{\mathrm{conv}}=1-2\varepsilon'$ fraction of insertions and any $1-\alpha_{C_1}-\varepsilon_{\mathrm{conv}}=1-\frac{\varepsilon}{46}-\varepsilon'=1-2\varepsilon'$ fraction of deletions in $O_{\varepsilon_1,\varepsilon_2}\left(N_1^{1+\varepsilon_0}+N_1\mathrm{poly}(\log N_1)\right)$.

We further take code $\tilde{C}_2$ from Corollary 3.6 with parameter $\varepsilon_{\tilde{C}_2}=2\varepsilon'$ and block length N_2 large enough so that $N_2\cdot q_{\tilde{C}_2}^{\varepsilon'/2\cdot N_2}\ge N_1$. $\tilde{C}_2$ is $\exp(\exp(\exp(N_2)))$-list decodable from any $1-2\varepsilon'$ fraction of insertions and $1-2\varepsilon'$ fraction of deletions.

String I for the indexing scheme is constructed according to the structure described in Section 5.1 using $\tilde{C}_1$ and $\tilde{C}_2$.

We define algorithm $\widetilde{\mathrm{ED}}_I$ as an algorithm that takes $S\times I$ and S' and runs an instance of Algorithm 2 with $S\times I$, S', N_1, N_2, and decoders of $\tilde{C}_i$ as its input. As codes $\tilde{C}_i$ list decode from $1-2\varepsilon'$ fraction of insertions and deletions, Lemma 6.4 guarantees that $\widetilde{\mathrm{ED}}_I$ generates a set of insertions and deletions over $S\times I$ that converts it to S' and is of size $(1+23\cdot 2\varepsilon')\mathrm{ED}(S\times I,S')+2\cdot\frac{\varepsilon_2}{2}|S'| = (1+\varepsilon)\mathrm{ED}(S\times I,S')+\varepsilon_2|S'|$ or less with probability $1-e^{-\frac{\varepsilon_2}{3N_1}} = 1-e^{-O\left(\frac{\varepsilon_2}{\log n}\right)} = 1-\frac{1}{n^{O_{\varepsilon_1,\varepsilon_2}(1)}}$.

Finally, since $N_1=O(\log n)$, $N_2=O(\log\log n)$, $\tilde{C}_1$ is list-decodable in $O(N_1^{1+\varepsilon_0}+N_1\mathrm{poly}(\log N_1))$ time and $\tilde{C}_2$ is efficiently list-decodable, Theorem 5.2 guarantees that $\widetilde{\mathrm{ED}}_I$ runs in

$$\begin{aligned}
&O_{\varepsilon_1,\varepsilon_2}\left(\frac{nT_{\mathrm{Dec}_{\tilde{C}_1}}(N_1)}{N_1}+\frac{nT_{\mathrm{Dec}_{\tilde{C}_2}}(N_2)}{N_2}+N_2L_1L_2\cdot n\log\log n\right)\\
=\;& O_{\varepsilon_1,\varepsilon_2}\left(\frac{n}{\log n}\cdot T_{\mathrm{Dec}_{\tilde{C}_1}}(\log n)+\frac{n}{\log\log n}\cdot T_{\mathrm{Dec}_{\tilde{C}_2}}(\log\log n)\right.\\
&\left.+n\log^2\log n L_{\tilde{C}_1}(N_1)L_{\tilde{C}_2}(N_2)\right)\\
=\;& O_{\varepsilon_1,\varepsilon_2}\left(n\log^{\varepsilon_0} n\right)
\end{aligned}$$

time. □

7 NEAR-LINEAR TIME INSDEL CODES

The construction of efficient (uniquely-decodable) insertion-deletion codes from [22] and list-decodable codes from [24] profoundly depend on decoding synchronization strings that are attached to codewords of an appropriately chosen Hamming-type code. The decoding procedure, which was introduced in [22], consists of multiple rounds of computing the longest common subsequence (LCS) between a synchronization string and a given string. In this section, we will show that using the indexing schemes that are introduced in this paper, one can compute approximations of the LCSs instead of exact LCSs to construct insertion-deletion codes of similar guarantees as in [22, 24] that have faster decoding complexity.

Specifically, for uniquely-decodable insertion-deletion codes, [22] provided codes with linear encoding-time and quadratic decoding-time that can approach the singleton bound, i.e., for any $0<\delta<1$ and $0<\varepsilon<1-\delta$ can correct from δ-fraction of insertions and deletions and achieve a rate of $1-\delta-\varepsilon$. Further, same authors [23] provided codes with linear encoding complexity and near-linear decoding complexity can can correct from $\delta<1/3$

fraction of insertions and deletions but only achieve a rate of $1-3\delta-\varepsilon$. In Theorem 2.4 we will provide insertion-deletion codes that give the best of the two worlds, i.e., approach the Singleton bound and can be decoded in near-linear time.

Further, in Theorem 7.4, we show that the same improvement can be made over list-decodable insertion-deletion codes of [24]. However, this improvement brings downs the complexity of all components of the decoding procedure to near-linear time except the part that depends on the decoding of a list-recoverable code that is used as a black-box in the construction from [24]. Even though this progress does not immediately improve the decoding time of list-decodable codes of [24], it opens the door to enhancement of the decoding complexity down to potentially a near-linear time by the future advances in the design of list-recoverable codes.

7.1 Enhanced Decoding of Synchronization Strings via Indexing

Synchronization strings, as introduced in [22], are strings that satisfy Definition 3.1. Let S be an ε-synchronization string that is communicated through a channel that suffers from a certain fraction of insertions and deletions. A decoding algorithm Dec_S for synchronization string S under such channel is an algorithm that takes string that is arrived at the receiving end of the channel, and for each symbol of that string, guesses its actual position in S. We measure the quality of the decoding algorithm Dec_S by a metric named as *misdecodings*. A misdecoding in the above-mentioned decoding procedure is a symbol of S that (1) is not deleted by the channel *and* (2) is not decoded correctly by Dec_S. (formal definitions in [22])

The important quality of synchronization strings that is used in the design of insertion-deletion codes in [22, 24] is that there are decoders for any ε-synchronization string that run in quadratic time $O(n^2/\varepsilon)$ and guarantee $O(n\sqrt{\varepsilon})$ misdecodings. In this paper, by indexing synchronization strings with indexing sequences introduced in Theorem 2.3, we will show that one can obtain a near-linear decoding that provides similar misdecoding guarantee.

In the rest of this section, we first present and prove a theorem that shows an indexed synchronization string can be decoded in near-linear time with guarantees that are expected in Theorem 6.14 of [22] and Lemma 3.2 of [24]. We then verify that the steps taken in [22, 24] still follow through.

Theorem 7.1. *Let S be a string of length n that consists of the coordinate-wise concatenation of an ε_s-synchronization string and an ε_I-indexing sequence from Theorem 2.3. Assume that S goes through a channel that might impose up to $\delta\cdot n$ deletions and $\gamma\cdot n$ symbol insertions on S for some $0\le\delta<1$ and $0\le\gamma$ and arrives as S' on the receiving end of the channel. For any positive integer K, there exists a decoding for S' that runs in $O(Kn\mathrm{poly}(\log n))$ time, guarantees up to $n\left(\frac{1+\gamma}{K(1+\varepsilon_I)}+\frac{\varepsilon_I(1+\gamma/2)}{1+\varepsilon_I}+K\varepsilon_s\right)$ misdecodings, and does not decode more than K received symbol to any number in $[1,n]$.*

Before proceeding to the proof of Theorem 7.1, we present and prove the following simple yet useful lemma.

Lemma 7.2. *Let us have a set of insertions and deletions that converts string S_1 to string S_2 which is of size $\mathrm{ED}_{APP}\le(1+\varepsilon)\mathrm{ED}(S_1,S_2)$. The common subsequence between S_1 and S_2 that is implied by such a set (LCS_{APP}) is of size $(1+\varepsilon)|\mathrm{LCS}|-\frac{\varepsilon}{2}(|S_1|+|S_2|)$ or larger.*

Proof.

$$\begin{aligned}|\mathrm{LCS}_{APP}| &= \frac{|S_1|+|S_2|-\mathrm{ED}_{APP}}{2}\\ &\ge \frac{|S_1|+|S_2|-(1+\varepsilon)\mathrm{ED}(S_1,S_2)}{2}\\ &= (1+\varepsilon)|\mathrm{LCS}|-\frac{\varepsilon}{2}(|S_1|+|S_2|)\end{aligned}$$

□

Proof of Theorem 7.1. The global decoding algorithm introduced in [22] and used in [24], consists of K repetitions of the following steps:

(1) Find the longest common subsequence (LCS) of S and S'.
(2) For any pair $(S[i],S'[j])$ in the LCS, decode $S'[j]$ as ith sent symbol.
(3) Remove all members of the LCS from S' (not in S).

Finally, the algorithm declares a special symbol $\perp$ as the decoded position of all elements of S' that are not included in any of the K LCSs.

To derive a decoding algorithm as promised in the statement of this lemma, we implement similar steps except we make use of the indexing scheme and compute an approximation of LCS instead of the LCS itself. This crucial step reduces the quadratic time required in the global decoding from [22] to near-linear time.

In [22], it has been shown that any assignment from Item 2 that is derived from any common subsequence between S and S' (not necessarily a LCS) does not contain more than $n\varepsilon_s$ misdecodings, i.e., successfully transmitted symbols of S that are decode incorrectly. (see 3.2). Therefore, after K repetitions, among symbols of S that are not deleted, there are at most $Kn\varepsilon_s$ ones that are decoded incorrectly.

To find an upper bound for the misdecodings of this algorithm, we need to bound above the number of successfully transmitted symbols that are not included in any LCS, i.e., decoded as $\perp$ as well. Let r be number of successfully transmitted symbols of S that remain undecoded after K repetitions of the matching procedure described above. Note that these symbols form a LCS of length r between S and the remainder of S' after all symbol eliminations throughout K repetitions. Indeed, this implies that the size of the LCS at the beginning of each repetition is at least r. Therefore, by Lemma 7.2, the size of the approximate longest common sequence found in each matching is at least $(1+\varepsilon_I)r-\varepsilon_I/2(|S|+|S'|)\ge(1+\varepsilon_I)r-\varepsilon_I n(1+\gamma/2)$. Note that sum of the size of all K common subsequences plus the remaining vertices cannot exceed $|S'|\le(1+\gamma)n$. Therefore,

$$\begin{aligned}&K\cdot[(1+\varepsilon_I)r-\varepsilon_I n(1+\gamma/2)]\le(1+\gamma)n\\ \Rightarrow\quad & r\le n\cdot\left[\frac{1+\gamma}{K(1+\varepsilon_I)}+\frac{\varepsilon_I(1+\gamma/2)}{1+\varepsilon_I}\right]\end{aligned}\tag{4}$$

Using (4) along with the fact that there are at most $Kn\varepsilon_s$ incorrectly decoded symbols of S gives that the overall number of misdecodings is at most $n\cdot\left[\frac{1+\gamma}{K(1+\varepsilon_I)}+\frac{\varepsilon_I(1+\gamma/2)}{1+\varepsilon_I}+K\varepsilon_s\right]$.

Further, as algorithm consists of K computations of the approximated longest common subsequence as described in Section 4, the running time complexity is $O(Kn\mathrm{poly}(\log n))$.

Finally, note that in each of the K rounds, there is at most one element that gets decoded as each number in $[1, n]$. Therefore, throughout the course of the algorithm, for each $i \in [1, n]$, there are at most K elements of S' that are decoded as i. □

7.2 Near-Linear Time (Uniquely-Decodable) Insertion-Deletion Codes

The construction of Singleton-bound-approaching uniquely-decodable insertion-deletion codes of [22] is consisted of a Singleton approaching error correcting code and a synchronization string. More precisely, for a given δ and ε and a sufficiently large n, [22] takes a synchronization string S of length n and a Singleton-bound-approaching error correcting code C with block length n (from [19]) and indexes each codeword of C, symbol by symbol, with symbols of S. If S is over alphabet Σ_S and C is over alphabet Σ_C, the resulting code would be over $\Sigma_C \times \Sigma_S$.

As for the decoding procedure, note that the input of the decoder is some code word of C, indexed with S, that might be altered by up to $\delta \cdot n$ insertions and deletions. Such insertions and deletions might remove some symbols, adds some new ones, or shift the position of some of them. The decoder uses the synchronization portion of each symbol to guess its actual position (in the codeword prior to $n \cdot \delta$ insertions and deletions) and then uses the decoder of code C to figure out the sent codeword.

Before proceeding to the proof of Theorem 2.4, we represent the following useful theorem from [22].

Theorem 7.3 (Implied by Theorem 4.2 from [22]). *Given a synchronization string S over alphabet Σ_S, an (efficient) decoding algorithm $\mathcal{D}_S$ with at most k misdecodings and decoding complexity $T_{\mathcal{D}_S}(n)$ and an (efficient) ECC C over alphabet Σ_C with rate R_C, encoding complexity $T_{\mathcal{E}_C}$, and decoding complexity $T_{\mathcal{D}_C}$ that corrects up to $n\delta + 2k$ half-errors, one obtains an insdel code that can be (efficiently) decoded from up to $n\delta$ insertions and deletions. The rate of this code is at least $\frac{R_C}{1+\log|\Sigma_S|/\log|\Sigma_C|}$. The encoding complexity remains $T_{\mathcal{E}_C}$, the decoding complexity is $T_{\mathcal{D}_C} + T_{\mathcal{D}_S}(n)$ and the complexity of constructing the code is the complexity of constructing C and S.*

We make use of Theorem 7.3 from [22] along with Theorem 7.1 to prove Theorem 2.4.

Proof of Theorem 2.4. As described earlier in this section, we construct this code by taking an error correcting code that approaches the Singleton bound and then index its codewords with symbols of an ε_s-synchronization string and an indexing scheme from Theorem 2.3 with parameter ε_I. For a given δ and ε, we choose $\varepsilon_I = \frac{\varepsilon}{18}, \varepsilon_s = \frac{\varepsilon^2}{288}$. Furthermore, we use the decoding algorithm from Theorem 7.1 with repetition parameter $K = \frac{24}{\varepsilon}$. With $\varepsilon_s, \varepsilon_I$, and K chosen as such, the decoding algorithm guarantees a misdecoding count of $n \cdot \left[\frac{1+\gamma}{K(1+\varepsilon_I)} + \frac{\varepsilon_I(1+\gamma/2)}{1+\varepsilon_I} + K\varepsilon_s\right] \le n \cdot \left[\frac{\varepsilon}{12} + \frac{\varepsilon}{12} + \frac{\varepsilon}{12}\right] = \frac{n\varepsilon}{4}$ or less. (note that there can be up to δn insertions, i.e., $\gamma \le \delta < 1$)

It has been shown in [23] that such synchronization string can be constructed in linear time over an alphabet of size $\varepsilon_s^{-O(1)}$. Also, the indexing sequence from Theorem 2.3 has an alphabet of size $\exp\left(\log(1/\varepsilon_I)/\varepsilon_I^3\right)$. Therefore, the alphabet size of the coordinate-wise concatenation of the ε_s-synchronization string and the indexing sequence is $|\Sigma_S| = \exp\left(\log(1/\varepsilon)/\varepsilon^3\right)$.

As the next step, we take code C from [19] as a code with distance $\delta_C = \delta + \frac{\varepsilon}{2}$ and rate $1 - \delta_C - \frac{\varepsilon}{4}$ over an alphabet of size $|\Sigma_C| = |\Sigma_S|^{4/\varepsilon}$. Note that $|\Sigma_S| = \exp\left(\log(1/\varepsilon)/\varepsilon^3\right)$, therefore, the choice of $|\Sigma_C|$ is large enough to satisfy the requirements of [19]. C is also encodable and decodable in linear time.

Plugging C and S as described above in Theorem 7.3 gives an insertion-deletion code that can be encoded in linear time, be decoded in $O(Kn\text{poly}(\log n))$ time, corrects from any δn insertions and deletions, achieves a rate of $\frac{R_C}{1+\log|\Sigma_S|/\log|\Sigma_C|} \ge \frac{1-\delta-3\varepsilon/4}{1+\varepsilon/4} \ge 1 - \delta - \varepsilon$, and is over an alphabet of size $\exp\left(\log(1/\varepsilon)/\varepsilon^4\right)$. □

7.3 Improved List-Decodable InsDel Codes

A very similar improvement is also applicable to the design of list-decodable insertion-deletion codes from [24] as it also utilizes indexed synchronization strings and a similar position recovery procedure. In the following theorem, we will provide a black-box conversion of a given list-recoverable code to a list-decodable insertion-deletion code that only adds a near-linear time overhead to the decoding complexity. Hence, the following theorem paves the way to obtaining insertion-deletion codes that are list-decodable in near-linear time upon the design of near-linear time list-recoverable codes. We will use the following theorem to prove Theorem 2.5 at the end of this section.

Theorem 7.4. *Let $C : \Sigma^{nR} \to \Sigma^n$ be a (α, l, L)-list recoverable code with rate R, encoding complexity $T_{Enc}(\cdot)$ and decoding complexity complexity $T_{Dec}(\cdot)$. For any $\varepsilon > 0$ and $\gamma \le \frac{l\varepsilon}{3} - 1$, by indexing codewords of C with an $\varepsilon_s = \frac{\varepsilon^2}{9(1+\gamma)}$-synchronization string over alphabet Σ_s and $\varepsilon_I = \frac{\varepsilon}{3(1+\gamma/2)}$-indexing sequence over alphabet Σ_I, one can obtain an L-list decodable insertion-deletion code $C' : \Sigma^{nR} \to [\Sigma \times \Sigma_s \times \Sigma_I]^n$ that corrects from $\delta < 1 - \alpha - \varepsilon$ fraction of deletions and γ fraction of insertions. C' is encodable and decodable in $O(T_{Enc}(n) + n)$ and $O_{\varepsilon,\gamma}(T_{Dec}(n) + n\text{poly}(\log n))$ time respectively.*

Proof. We closely follow the proof of Theorem 3.1 from [24] except that we use an indexed synchronization string to speed up the decoding procedure.

Index the code C with an $\varepsilon_s = \frac{\varepsilon^2}{9(1+\gamma)}$-synchronization string and an $\varepsilon_I = \frac{\varepsilon}{3(1+\gamma/2)}$-indexing sequence as constructed in Theorem 2.3 to obtain code C'.

In the decoding procedure, for a given word $\tilde{x}$ that is δn deletions and γn insertions far from some codeword $x \in C'$, we first use the decoding algorithm from Theorem 7.1 to decode the index portion of symbols with parameter $K = 3(1+\gamma)/\varepsilon$. This will give a list of up to $K = 3(1+\gamma)/\varepsilon \le l$ candidate symbols for each position of the codeword x.

We know from Theorem 7.1 that all but

$$n\left(\frac{1+\gamma}{K(1+\varepsilon_I)} + \frac{\varepsilon_I(1+\gamma/2)}{1+\varepsilon_I} + K\varepsilon_s\right) \le n\left(\frac{\varepsilon}{3(1+\varepsilon_I)} \times 2 + \frac{\varepsilon}{3}\right) \le n\varepsilon$$

of the symbols of x that are not deleted are in the correct list. As there are up to $n(1-\delta)$ deleted symbols, all but $n(1-\delta-\varepsilon) > n\alpha$ of the lists contain the symbol from the corresponding position in x. Having such lists, the receiver can use the list-recovery function of C to obtain an L-list-decoding for C'.

The encoding complexity follows from the fact that synchronization strings be constructed in linear time [15, 23], the decoding complexity follows from Theorem 7.1, and the alphabet of C' is trivially $\Sigma \times \Sigma_s \times \Sigma_I$ as it is obtained by indexing codewords of C with the ε_s-synchronization string and the ε_I-indexing sequence. □

We now use Theorem 7.4 to prove Theorem 2.5.

Proof of Theorem 2.5. Take list-recoverable code C from Theorem 6.2 with parameters $\rho_C = 1 - \delta - \frac{\varepsilon}{2}$, $\varepsilon_C = \frac{\varepsilon}{4}$, $l_C = \frac{12\gamma+4}{\varepsilon}$, and $c_C = \varepsilon_0$ over an alphabet Σ of adequately large size $|\Sigma| \geq q_{0,C}$ which we determine later. According to Theorem 6.2, C has a rate of ρ_C and a randomized $(\rho_C + \varepsilon_C, l_C, L(n) = \exp(\exp(\exp(\log^* n))))$-list recovery that works in $O(n^{1+\varepsilon_0})$ time and succeeds with probability 2/3.

We plug code C into Theorem 7.4 with parameters $\varepsilon_{\text{conv}} = \frac{\varepsilon}{4}$ and $\gamma_{\text{conv}} = \gamma$ to obtain code C'. We can do this because $\gamma_{\text{conv}} \leq \frac{l_C \varepsilon_{conv}}{3} - 1$. According to Theorem 7.4, C' is $L(n)$-list decodable from $1 - \rho_C - \varepsilon_C - \varepsilon_{\text{conv}} = \delta$ fraction of deletions and γ fraction of insertions in $O(n^{1+\varepsilon_0})$. This list-decoding is correctly done if the list-recovery algorithm works correctly. Therefore, the list decoder succeeds with probability 2/3 or more.

Note that the ε_s-synchronization strings in Theorem 7.4 exist over alphabets of size $|\Sigma_s| = \varepsilon_s^{-O(1)}$ and ε_I-indexing sequence exist over alphabets of size $|\Sigma_I| = \exp\left(\log(1/\varepsilon_I)/\varepsilon_I^3\right)$. Therefore, if we take alphabet Σ large enough so that $|\Sigma| \geq \max\left\{|\Sigma_s \times \Sigma_I|^{2/\varepsilon}, q_{0,C}\right\} = \max\left\{\exp\left(\log(1/\varepsilon)/\varepsilon^4\right), q_{0,C}\right\} = O_{\varepsilon_0,\varepsilon,\gamma}(1)$ the rate of the resulting code will be $\frac{\rho_C}{1+\log(|\Sigma_S|\times|\Sigma_I|)/\log|\Sigma|} \geq \frac{1-\delta-\varepsilon/2}{1+\varepsilon/2} \geq 1-\delta-\varepsilon$.

Finally, the encoding and decoding complexities directly follow from Theorem 7.4. □

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight Hardness Results for LCS and Other Sequence Similarity Measures. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*. 59–78.

[2] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. 2016. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*. 375–388.

[3] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. 2014. Consequences of Faster Alignment of Sequences. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*. 39–51.

[4] Noga Alon, Jeff Edmonds, and Michael Luby. 1995. Linear time erasure codes with nearly optimal recovery. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 512–519.

[5] Alexandr Andoni and Robert Krauthgamer. 2008. The Smoothed Complexity of Edit Distance. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*. 357–369.

[6] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. 2010. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 377–386.

[7] Alexandr Andoni and Krzysztof Onak. 2012. Approximating edit distance in near-linear time. *SIAM J. Comput.* 41, 6 (2012), 1635–1648.

[8] Arturs Backurs and Piotr Indyk. 2015. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 51–58.

[9] Ziv Bar-Yossef, TS Jayram, Robert Krauthgamer, and Ravi Kumar. 2004. Approximating edit distance efficiently. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*. IEEE, 550–559.

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

[11] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. 2018. Approximating Edit Distance in Truly Subquadratic Time: Quantum and MapReduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1170–1189.

[12] Karl Bringmann and Marvin Künnemann. 2015. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*. 79–97.

[13] Karl Bringmann and Marvin Künnemann. 2018. Multivariate Fine-Grained Complexity of Longest Common Subsequence. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1216–1235.

[14] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. 2018. Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*. 979–990.

[15] Kuan Cheng, Bernhard Haeupler, Xin Li, Amirbehshad Shahrasbi, and Ke Wu. 2019. Synchronization Strings: Efficient and Fast Deterministic Constructions over Small Alphabets. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2185–2204.

[16] Václav Chvátal, David A. Klarner, and Donald E. Knuth. 1972. *Selected combinatorial research problems.* Technical Report. Computer Science Department, Stanford University.

[17] Pawel Gawrychowski. 2012. Faster Algorithm for Computing the Edit Distance between SLP-Compressed Strings. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*. 229–236.

[18] Shafi Goldwasser and Dhiraj Holden. 2017. The Complexity of Problems in P Given Correlated Instances. In *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*. 13:1–13:19.

[19] Venkatesan Guruswami and Piotr Indyk. 2005. Linear-time encodable/decodable codes with near-optimal rate. *IEEE Transactions on Information Theory* 51, 10 (2005), 3393–3400.

[20] Venkatesan Guruswami and Ray Li. 2016. Efficiently decodable insertion/deletion codes for high-noise and high-rate regimes. In *Information Theory (ISIT), 2016 IEEE International Symposium on*. IEEE, 620–624.

[21] Venkatesan Guruswami and Carol Wang. 2017. Deletion codes in the high-noise and high-rate regimes. *IEEE Transactions on Information Theory* 63, 4 (2017), 1961–1970.

[22] Bernhard Haeupler and Amirbehshad Shahrasbi. 2017. Synchronization Strings: Codes for Insertions and Deletions Approaching the Singleton Bound. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*.

[23] Bernhard Haeupler and Amirbehshad Shahrasbi. 2018. Synchronization strings: explicit constructions, local decoding, and applications. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*. 841–854.

[24] Bernhard Haeupler, Amirbehshad Shahrasbi, and Madhu Sudan. 2018. Synchronization Strings: List Decoding for Insertions and Deletions. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*.

[25] Bernhard Haeupler, Amirbehshad Shahrasbi, and Ellen Vitercik. 2018. Synchronization Strings: Channel Simulations and Interactive Coding for Insertions and Deletions. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*. 75:1–75:14.

[26] Brett Hemenway, Noga Ron-Zewi, and Mary Wootters. 2017. Local List Recovery of High-rate Tensor Codes and Applications. *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)* (2017).

[27] Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (1977), 664–675.

[28] James W Hunt and Thomas G Szymanski. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM* 20, 5 (1977), 350–353.

[29] William Kuszmaul. 2019. Efficiently Approximating Edit Distance Between Pseudorandom Strings. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1165–1180.

[30] Gad M Landau, Eugene W Myers, and Jeanette P Schmidt. 1998. Incremental string comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582.

[31] Vladimir Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR 163* 4 (1965), 845–848.

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

[33] Aviad Rubinstein. 2018. Approximating Edit Distance. https://theorydish.blog/2018/07/20/approximating-edit-distance/.

[34] Leonard J. Schulman and David Zuckerman. 1999. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE transactions on information theory* 45, 7 (1999), 2552–2557.

[35] Michael Sipser and Daniel A Spielman. 1994. Expander codes. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. IEEE, 566–576.

[36] Daniel A Spielman. 1996. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory* 42, 6 (1996), 1723–1731.

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