1 Overview

In this lecture, we discuss:

1. motivation/definitions of synchronization strings and related ideas.
2. an efficient encoding/decoding procedure that handles synchronization errors.
3. construction of synchronization strings.

2 Review

2.1 Motivation

Error-correcting codes are information-efficient protocols for parties to communicate across noisy channels. Traditionally, the channel is assumed to introduce a bounded number of Hamming errors or erasures, and these errors are either random or adversarial. Standard techniques however typically cannot handle “synchronization errors” that cause the transmitted and received strings to be misaligned (e.g. insertion/deletion, a.k.a. ins/del errors). Synchronization strings introduced in [HS17a, HS17b, CHL+18] are an efficient method to deal with such errors. We refer the reader to the original text for a fuller picture.

2.2 A naive attempt

Recall the failed approach from last class. If we append to each sent symbol that symbol’s index in the message, we argued last time that we may treat ins/del errors as Hamming errors, and reduce the problem to error correction for the Hamming channel.

\[
X_1 \rightarrow (X_1, s_1)
\]
\[
X_2 \rightarrow (X_2, s_2)
\]
\[
\vdots
\]
\[
X_n \rightarrow (X_n, s_n)
\]

Unfortunately, we need a super-constant sized alphabet to encode the index, which is not acceptable in practice. Instead, we will pick a string from a constant sized alphabet to send with the message that can still be used to recover the alignment.

3 Synchronization Strings

Rather than use a string with a lot of structure (e.g. repeating only so often) which can often be exploited by an adversary picking the ins/del errors, we will use strings that are nowhere self-similar. Perhaps unsurprisingly since this is a class about the power of randomization, we will show that random strings make for good candidates.

Given a string \(s_1, s_2, \ldots, s_n\), we think of the prefix of the first \(i\) symbols as an encoding of the number \(n\). Formally, we write \(C_S = \{S[i, i] \mid i \in \{1, \ldots n\}\}\) to denote the “code” of \(S\).
3.1 Edit Distance

If we are thinking of prefixes of $S$ as a code, we will need to argue that a small number of insertions and deletions cannot force us to confuse one codeword for another. What is a suitable metric in which to think of prefixes of a string as codewords? A first and natural idea is the following:

**Definition 1** (Edit Distance). The edit distance between strings $A$ and $B$ is the function:

$$ED(A, B) := \text{minimum number of ins/dels needed to transform } A \rightarrow B$$

**Exercise 1.** Show that edit distance is a metric.

**Exercise 2.** Show that $ED(A, B) = |A| + |B| - 2 \cdot LCS(A, B)$, where $LCS(A, B)$ is the length of the longest common subsequence of $A$ and $B$.

3.2 Relative Suffix Distance

Unfortunately this is not really the right idea since the edit distance between $S[1, i]$ and $S[1, i-1]$ is small. Intuitively, there is an unavoidable sensitivity to recent corruptions, so a better notion is one that weights errors by their recency:

**Definition 2** (Relative Suffix Distance). The relative suffix distance between strings $A$ and $B$ is the function:

$$RSD(A, B) := \max_{k > 0} \frac{ED(A[|A| - k, |A|], B[|B| - k, |B|])}{2k}$$

**Claim 3.** $RSD$ is a metric.

**Proof.** $RSD$ is clearly symmetric, is clearly nonnegative, and strictly positive iff $A \neq B$. To show that the triangle inequality holds, fix three strings $S_1$, $S_2$, and $S_3$, and let $k_{ij}$ be the $k$ maximizing the ratio in the definition of $RSD(S_i, S_j)$. Then:

$$RSD(S_1, S_3) = \frac{ED(S_1[|S_1| - k_{13}, |S_1|], S_3[|S_3| - k_{13}, |S_3|])}{2k_{13}} \leq \frac{ED(S_1[|S_1| - k_{13}, |S_1|], S_2[|S_2| - k_{13}, |S_2|])}{2k_{13}} + \frac{ED(S_2[|S_2| - k_{13}, |S_2|], S_3[|S_3| - k_{13}, |S_3|])}{2k_{13}}$$

(by since $ED$ is a metric)

$$\leq \frac{ED(S_1[|S_1| - k_{12}, |S_1|], S_2[|S_2| - k_{12}, |S_2|])}{2k_{12}} + \frac{ED(S_2[|S_2| - k_{23}, |S_2|], S_3[|S_3| - k_{23}, |S_3|])}{2k_{23}}$$

(by $k_{ij}$ is the maximizer in $RSD(S_i, S_j)$)

$$= RSD(S_1, S_2) + RSD(S_2, S_3)$$

The hope is that now we can construct a codeword such that its own prefixes are far away in $RSD$.

3.3 Notions of self-similarity

Armed with the right metric, we can define what it means for a string to resemble itself:

**Definition 3** ($\epsilon$-synchronization string). $S$ is an $\epsilon$-synchronization string if for all indices $i < j < k$:

$$ED(S(i, j), S(j, k)) \geq (1 - \epsilon) \cdot (k - i)$$

Note that up to relabeling, the only 0-synchronization string is the string $1, 2, 3, \ldots n$. 

11-2
Lemma 4. If \( S \) is an \( \epsilon \)-synchronization string, then \( C_S \) has RSD distance at least \( 1 - \epsilon \) to any of its prefixes.

Proof. For any indices \( \ell \) and \( \ell' \) such that \( \ell' > \ell \), by definition:

\[
ED(S[2\ell - \ell', \ell], S[\ell, \ell']) \geq (1 - \epsilon)2(\ell' - \ell)
\]

Thus for any prefix \( S' \) of \( S \), it holds that \( RSD(S, S') \geq (1 - \epsilon)\frac{2(\ell' - \ell)}{2(\ell' - \ell)} = (1 - \epsilon). \)

As an aside:

Puzzle 1. Can one construct an infinitely long string such that any two neighboring strings are different? If so what is the minimum alphabet size necessary (clearly 2 is not enough)?

Solution: The following construction works. Construct a string \( S \) such that the \( i \)th symbol is the parity of the binary expansion of \( i \). Now construct \( S' \) by letting the \( i \)th symbol be \( S(i) - S(i + 1) \). The result is a string on the alphabet \( \{0, 1, -1\} \) with the desired properties. The proof of correctness is involved and we omit it...

We may also define the following weaker version of self-similarity. Define a monotone matching between two strings to be a common subsequence to them.

Definition 4 (\( \epsilon \)-self matching string). A string \( S \) of length \( n \) is an \( \epsilon \)-matching string if any monotone matching between \( S \) and itself matches strictly less than \( \epsilon n \) pairs that do not correspond to the same element of the string.

As an example, the solid edges correspond to “good” edges between same elements of the string, the dashed edges correspond to “bad” edges. No more than \( \epsilon n \) “bad” edges are allowed in an \( \epsilon \)-self matching string.

![Diagram of self-similarity example]

We note that if instead we required that \#bad < \( \epsilon(n - \#good) \), this would be equivalent to the \( \epsilon \)-synchronization property up to a factor of 2. To see one direction, suppose this property is satisfied. Fix any indices \( i < j < k \), and consider the monotone matching between \( S \) and itself that matches all characters before \( i \) and after \( k \) to themselves with “good” edges, and then has “bad” edges between members of the LCS of \( S[i, j] \) and \( S[j, k] \). By the identity \( ED(A, B) = |A| + |B| - 2 \text{LCS}(A, B) \), it follows that the edit distance between \( S[i, j] \) and \( S[j, k] \) is lower bounded by \((1 - 2\epsilon)(k - i)\).

For a proof sketch of the other direction, suppose \( S \) is an \( \epsilon \)-synchronization string. Given a monotone matching between \( S \) and itself, one can partition the “bad” edges into groups such that within each group with the following property. The left endpoints of these edges lie in an interval \( A \), the right endpoints of these edges lie in an interval \( B \), and \( A \) and \( B \) are disjoint. Since \( S \) satisfies the \( \epsilon \)-synchronization property, \( \text{LCS}(A, B) < (|A| + |B| - (1 - \epsilon)(|A| + |B|))/2 = \epsilon(|A| + |B|)/2. \)
Summing over all groups in the partition yields the claim. See Theorem 6.2 of [HS17a] for more explicit details.

Also note that unlike the $\epsilon$-synchronization property, the $\epsilon$-self matching property is not hereditary.

### 3.4 Decoding Procedure

Before demonstrating the existence of non-self similar strings, we first show how to use them in practice by giving a decoding procedure. In what follows, we assume that $S$ is $\epsilon$-self matching.

<table>
<thead>
<tr>
<th>Received</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\hat{X}_1, \hat{S}_1)$</td>
<td>$(X_1, S_1)$</td>
</tr>
<tr>
<td>$(\hat{X}_2, \hat{S}_2)$</td>
<td>$(X_2, S_2)$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$(\hat{X}_n, \hat{S}_n)$</td>
<td>$(X_n, S_n)$</td>
</tr>
</tbody>
</table>

**Algorithm 1 Decoding Procedure**

```plaintext
for $i = 1 \ldots \epsilon^{-1/2}$ do
    Compute $M_i \leftarrow LCS(\hat{S}, S)$.
    Remove $M_i$ from $\hat{S}$.
end for

Let the final matching be $M \leftarrow M_1 + M_2 + \ldots + M_{\epsilon^{-1/2}}$.
Compute $\hat{X}$ based on $M$ as follows: if a position $i$ has exactly one match, copy in symbol $\hat{X}_i$, otherwise use "?" symbol.

return $\hat{X}$.
```

To analyze the number of extra errors compared to the trivial numbering scheme, we define two types of errors.

- **Type I**: Correctly received symbol matched incorrectly.
- **Type II**: Correctly received symbol not matched at all.

Note that incorrectly received symbols are counted as errors in the trivial numbering scheme as well, so we do not need to count them.

How many Type I errors can we possibly have? At most $\epsilon n!$! For the following reason. Every Type I error symbol was correctly received, and so came from a symbol $A$. On the other hand it was matched to a symbol $B$. Thus $A - B$ is a “bad” edge in the monotone matching induced by the decoding. Since $S$ is chosen to be $\epsilon$-self matching, there can be at most $\epsilon n$ such errors.

What about Type II errors? If there were $\epsilon$ many Type II errors, this would imply that every matching $M_i$ in the algorithm has $\geq \epsilon$ matches. This would mean that the total number of matches is at least $\epsilon/\sqrt{\epsilon}$. Since the total number of corruptions has to be less than $2n$, we deduce that:

$$\frac{\epsilon}{\sqrt{\epsilon}} < 2n$$

$$\Rightarrow \epsilon < 2n \sqrt{\epsilon}$$

Thus the total number of extra errors is strictly less than $3n \sqrt{\epsilon}$. From here, one can use any Hamming/erasure channel error correcting code to fix the remaining errors.
Note that the decoding procedure gives an additive error guarantee. Usually one can pick $\epsilon$ such that this is not a problem. It is actually also possible to obtain a multiplicative guarantee, but for that we need a more involved decoding procedure based on $RSD$. Incidentally, that one has the nice property that it can be performed online!

### 3.5 Construction

The last remaining question is how do we construct $\epsilon$-self matching strings in $O(1)$ alphabet size? Answer: random strings!

**Observation 5.** If $S$ is from a $q$-ary alphabet, there is a self matching of size at least $n/q - 1$.

**Observation 6.** For a random $S$, the expected maximal self matching is at least of length $\Omega\left(\frac{n}{\sqrt{q}}\right)$.

To see it is also $O\left(\frac{n}{\sqrt{q}}\right)$, we use a union bound over all bad events in which a subsequence of length $\epsilon n$ is matched to itself.

The probability a matching of length $\ell$ is valid is $q^{-\ell}$. If $\ell = \epsilon n$, the number of possible matchings is $\left(\frac{1}{q^{\ell}}\right)^2 \leq \left(\frac{1}{q^{\epsilon n}}\right)^2 = \left(\frac{1}{q^{\sqrt{q}}}\right)^{2\epsilon n}$. Thus a random $S$ over alphabet of size $O(\epsilon^{-2})$ is an $\epsilon$-self matching string with exponentially high probability.

We end by noting that if instead we want an $\epsilon$-synchronization string, we can reuse our technique for performing non-repetitive coloring for line graphs. Punchline: the LLL! If intervals do not overlap, their bad events are independent, otherwise the probability that they have a large matching is the same calculation as above.

### 4 Conclusion

In this lecture we discussed various definitions relating to synchronization strings and how they relate to error correcting codes in the face of ins/del channels. We saw at least one explicit decoding procedure demonstrating how to use these in practice which gives an additive bound on the extra number of errors introduced. Finally we gave a simple construction of synchronization strings using randomization.

### References

