CS 229r Essential Coding Theory April 29, 2020

Lecture 25

Instructor: Madhu Sudan *Scribe: Aditya Dhar*

1 Outline

- Coding for editing errors
- Recap of course

2 Codes for Editing Errors

2.1 Definitions

We have two strings $X \in \Sigma^n$ and $Y \in \Sigma^m$.

Definition 1 (X$\rightarrow_{\Delta,\Gamma} Y$)**.** Deleting $\leq \Delta$ symbols from X and inserting $\leq \Gamma$ symbols gives Y.

We want to design a $(\Delta, \Gamma)-$edit distance code:

- Unique decoding: $D(Y) = X$
  - Encoding function, E, maps $\Sigma^k \rightarrow \Sigma^n$
  - Decoding function, D, maps $\Sigma^* \rightarrow \Sigma^k$, unlike other decoders, where we map from Σ^n.
- List decoding: $X \in D(Y)$.
  - Use the same encoding function, E, maps $\Sigma^k \rightarrow \Sigma^n$
  - Use a different decoding function, D, maps $\Sigma^* \rightarrow \binom{\Sigma^k}{L}$. In this case, we don't ask the decoding function to output just the message; the decoding function returns an output that includes the desired message.

$\forall X \in \Sigma^k$, we find $Y \in \Sigma^*$ such that $E(X) \rightarrow_{\Delta,\Gamma} Y$.
Thus, given a family of code $E = (E_n : \Sigma^{k_n} \rightarrow \Sigma^n)_n$, with rate $R = \lim_{n\rightarrow\infty} \frac{k_n}{n}$, we can construct a (δ, γ)-code if E_n is a $(\delta n, \gamma n)$-code $\forall n$.
Question: what values of δ, γ, R are achievable?

2.2 Background

Schulman and Zuckerman '99 first proposed an insertion / deletion editing code over $q = poly(n)$ and a rate of $R \rightarrow 1 - (\delta + \gamma)$. The proof was algorithmic and proved unique decoding. For an initial thought experiment to see why rate is bounded at $1 - (\delta + \gamma)$, we examine a pigeonhole principle-ish proof:

- Take a code of rate R. The usual pigeonhole argument says there are two codewords that agree on the first Rn coordinates, and the rest of the codes are completely different strings.
- Assume $R = 1 - (\delta + \gamma)$

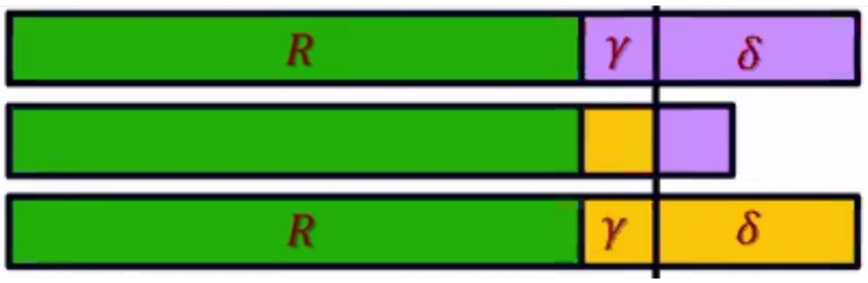

Figure 1: Construction of the intermediate codeword

- Produce an intermediate word, by copying the Rn coordinates over, deleting the delta coordinates from the bottom and adding the gamma coordinates from the top. We can also insert the gamma coordinates from the bottom and then add the delta coordinates from the top. In either case, we have a delta deletion, gamma insertion, which we can do with any choice of gamma or delta.
- We can only do this when q is very large, giving us the $poly(n)$ size.

How was the actual code constructed? Take a Hamming code $E : \Sigma^k \to \Sigma^n \to E' : \Sigma^k \to (\Sigma \times [n])^n$

- Define $E'(x)_i = (E(x)_i, i)$
- Rate(E') = Rate$(E) - \frac{\log n}{\log |\Sigma|}$ for $n << |\Sigma|$

So, for some Hamming code, we construct an edit code E' where the new encoding of x in the $i-$th location is a pair including the index i. This new alphabet is larger and takes pairs; which is convenient when we are concerned about editing errors; each symbol also contains the location of the deletion. This works because we can rewrite an insertion or deletion for E' as an erasure in E; and a same-location insertion/deletion (the adversary deletes a coordinate in location i and then inserts another coordinate at location i as an error in E (two errors that translate into a single Hamming error). So, if we can deal with a certain number of insertions and deletions with a code E of distance $\gamma + \delta$, we can construct a $(\delta, \gamma)-$edit distance code. The best code of distance $\gamma + \delta$ is a Reed-Solomon code, with rate $1 - (\delta + \gamma)$; so the resulting edit code will be arbitrarily close, losing a little bit by the log terms.

2.3 Recent Work

Schulman-Zuckerman only works when q is very large. We want to think about constant-q list deocding, where $q = O(1)$. This does not mean that q is binary, but that it is a very large constant. In order to do this, we need something similar to indexing, but we don't want to use an alphabet growing with n.

- *Haeupler-Shahrasbi '17:* For $q = O(q)$, $R \to 1 - (\delta + \gamma)$. This is the same rate as Schulman and Zuckerman, also for unique decoding. This effectively simulates a Hamming error: deletion of a character and insertion in the same location. So, $HS'17$ implies $R = 1 - 2\delta$, subsuming Hamming bounds and approaching the singleton bound.
- *Haeupler-Shahrasbi-Sudan '18:* For $q = O_{\gamma,\delta,R}(1)$, $R \to 1 - \delta$. There is no restriction on γ, so while list size may grow in γ with adversarial addition of a significant number of symbols, we can still output the original message - despite large γ! This work subsumes what we know for list decoding, and is what we have seen in a previous problem set.

Haeupler-Shahrasbi strategy:

- Synchronization string $S = S_i \in [c]^n$ with a constant sized set $c = O(1) << n$

- We still have $E : \Sigma^k \to \Sigma^n$, but we construct off of this an edit-code $E' : \Sigma^k \to (\Sigma \times [c])^n$
- Our new edit code is constructed as $E'(x)_i = (E(x)_i, S_i)$
- $\text{Rate}(E') = \text{Rate}(E) - \frac{\log c}{\log |\Sigma|}$

This is fairly similar to the original Schulman-Zuckerman scheme, except we no longer have that $i \neq j \implies S_i \neq S_j$; we can't hope to simulate this with $c < n$ by the pigeonhole principle. So, we can define properties of S that are useful, presuming that S is constructible (as a tamer task than constructing a 2^k error correcting code).

Definition 2 (Synchronization Strings)**.** A string is a $\varepsilon-$synch. string if $\forall S-$ matching M, $|M| \leq \varepsilon n.$, where we call $M = \{(i_t, j_t)|1 \leq t \leq m\} \subseteq [n] \times [n]$ $S-$matching if

- there are distinct $i_1, ..., i_m$, $j_1, \ldots, j_m$
- This is nontrivially $S-$valid: $S_{i_t} = S_{j_t}$ but $i_t \neq j_t$ $\forall t$. There should be no vertical matching, and an element should be matched always to the same letter.
- This is monotone: $i_a < i_b \implies j_a < j_b$, thus maintaining the relative sequencing of elements.

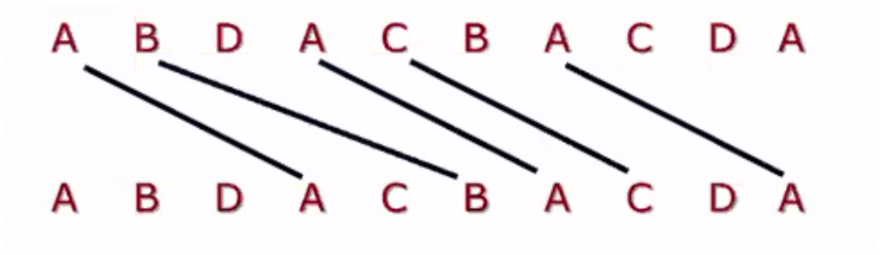

Figure 2: An example of a matching preserving monotonicity with no trivial matching.

We can verify if S is ε-synch. in polynomial time using dynamic programming to check what the best string is to match. So, for all $\varepsilon > 0$, $\exists c < \infty$ such that $\forall n \exists S \in [c]^n$ constant edit distance codes that are rate optimal. The relevant question is then how said strings help decoding errors.

Exercise 3. *Prove that a random string is $\varepsilon-$synch. w.h.p.*

3 List decoding + Synchronization Strings

3.1 Definitions

- List decodability: $C \subseteq \Sigma^n$ is (δ, L)-list-decodable if
  - for every $y = (y_1, ..., y_n) \in \Sigma^n$ if $S = \{x \in C | \#\{i | x_i \neq y_i\} \leq \delta n\}$, then $|S| \leq L$ and S can be computed efficiently given y.

  Exercise 4. *Prove that for any q and $\gamma < q - 1$, the family of codes with rate $R < 1 - \log_q(\gamma + 1) - \gamma \log_q \frac{\gamma+1}{\gamma} - \frac{\gamma+1}{l+1}$ is list-decodable with a $l-$sized list from any γn insertions w.h.p.*

- List-recoverable codes:
  - $C \subseteq \Sigma^n$ is (l, δ, L) list recoverable if: for every $Y_1 \times ... Y_n \subseteq \Sigma^n$ s.t. $|Y_i| \leq l$, if the set of codewords in C where $x_i \notin Y_i$ has size less than L: $S = \{x \in C | \#\{i | x_i \notin Y_i\} \leq \delta n\}$.

  - A starting point for constructing this is a good list decodable code; except instead of being a single word for each coordinate, we're given a set of values - the relevant question is to find codewords which pass through many of these sets, given that we miss a $\delta-$fraction (errors).

Theorem 5 (Guruswami-Rudra'06). *$\forall l, \delta, \varepsilon > 0$, $\exists$ Σ, L such that $\exists$ a family of (l, δ, L)-list-recoverable codes of rate $1 - \delta - \varepsilon$*

- So, regardless of how large the l is, we can make a list-recoverable code of rate $1 - \delta - \varepsilon$, with possible starting points being Folded RS codes or the Guruswami-Indyk alphabet reduction.

3.2 Edit Errors

Theorem 6. *Let $E : \Sigma^k \to \Sigma^n$ be $(l, \delta + \varepsilon, L)$-list-recoverable. Let $S \in [c]^n$ be an $\varepsilon'-$synch. string. Then $E' : \Sigma^k \to (\Sigma \times [c])^n$ given by $E'(x)_i = (E(x)_i, S_i)$ is a $(\delta, \gamma)-$list-decodable code.*

- *Where $l = \frac{2(1+\gamma)}{\varepsilon}$ and $\varepsilon' = \frac{\varepsilon}{2l}$*

We write an algorithm to prove that this works: given $(a_i, b_i)_{i \in [m]}, a_i \in \Sigma, b_I \in [c]$,

- Set $B = (b_1, \dots b_m); Y_1 = \cdots = Y_n = \varnothing$. We want to write $S = s_1, \dots, s_n$-monotone matching without crossovers, finding the largest such matching. This problem is solvable in polynomial time, because it is just the largest common subsequence problem
- For l iterations, do:
  - Let M be the largest monotone matching between B and S.
  - Remove the matched part from B, add matched α-symbols into $Y_i S$: $b_{\leftrightarrow} S_i \implies Y_i \leftarrow Y_i \cup \{a_j\}$. *Note:* At each iteration i, there is at most one element added to Y_i, because b_j can only be mapped to one s_i per iteration, and is then deleted and consequently cannot be mapped again.
  - List recover from $Y_1, \dots, Y_n$. These are subsets of Σ of at most size l, because we add one element per iteration over l iterations.

3.3 Analysis of Algorithm

Say that we have transmitted $E'(x)$, received pairs $(a_j, b_j)_{j \in [m]}$. If we can make sure that the i-th symbol encoding $E(x)_i$ does not appear in Y_i for at most $\delta + \varepsilon$-fraction of the coordinates, we can use list-recoverable codes from there.

- There can be δn errors from deletions: if $E'(x)$ was transmitted, and the adversary deleted some δn coordinates, there are at least $\delta-$fraction errors because that encoding will not appear in Y_i.
- The other errors happen if $E'(x)_i = (a_j, b_j)$ is not deleted, but $E(x)_i \notin Y_i$. There are two possible cases here:
  - Case 1: b_j is matched to S_i for $i' \neq i$:
    * Because we have an $\varepsilon'-$synch. string, this implies that there are at most $\varepsilon' n$ errors per iteration, because we start in location i, went to location j, and then went to location i'. These two characters are the same and this transformation is monotone. So, there can be at most $\varepsilon' n$ errors from insertion.
    * Over l iterations, this means we have at most $l\varepsilon' n$ errors $\implies \leq \frac{\varepsilon n}{2}$ errors.
  - Case 2: b_j remains unmatched at the end:

    * Say that αn code symbols are unmatched at the end. By construction, each iteration must match $\geq \alpha n$ symbols, else there would be a longest common subsequence of length αn (*Note: because E' has a perfect matching with S, the αn unmatched symbols are necessarily left intact in sequence*). Over l rounds, this gives us $l\alpha n \leq m \leq (1+\gamma(n)) \implies \alpha n \leq \frac{(1+\gamma)n}{l} \leq \frac{\varepsilon n}{2}$

This gives us at most εn errors from insertion and δn errors fromd deletion, so we know that at most $\delta + \varepsilon-$fraction of errors are corrupted, and we can then use list recoverable codes.

Further work being done on this problem includes editing and interaction errors, binary codes for edit distance (see Guruswami, Haeupler, Shahrasbi '19), and coding on small alphabets rather than large ones.

4 Course Recap!

What did we do?

- Existence and limits of codes:
  - Greedy and random constructions, treated as largely interchangeable.
  - How do you prove that codes of a certain type don't exist? Packing bounds, list packing, embedding into Euclidean space: there are codes that shouldn't touch; codes for which the Hamming balls overlap (but not by much), and codes of large distances cannot have too many code words. Embedding into Euclidean space is especially useful by embedding from Hamming space and then allowing us to use analysis over real vectors.
- Constructions + Algorithms:
  - Algebraic codes: these have very nice combinatorial properties that are mostly better than random. The only code where we didn't cover anything explicit was Reed-Muller, which seems to do worse than random codes. However, RM codes are locally decodable and testable while random codes are not.
  - Graph theoretic and information theoretic (polar) codes, where the motivation for these is algorithmic, achieving what we can get that doesn't work in other codes.
- Advanced topics
  - Local decoding
  - Interactive coding - rarely used in practice, most interactive protocols have a constant number of rounds of interaction; in which case you might as well encode each round of communication
  - Edit Errors - today's lecture!
- Applications in complexity:
  - Short coverage of topics in 'baby' pseudorandomness - limited independence and $\varepsilon-$bias - and 'baby' crypotgraphy - hard code bias

What didn't we do:

- Bounds: Cover linear programming bounds that tell us that Chernoff bounds are largely optimal
- Codes: Didn't give explicit constructions for algebraic-geometric codes
- Algorithms:
  - Low density parity check codes and decoding. Luckily, these are dominated by polar codes, so there are no nice theorems we'd learn with covering LDPC codes.

  - Linear time machinery (see Spielman codes, Guruswami and Indyk '03 for further reference)

- Explicit graph construction - despite using many for samplers, expanders, etc.
- "Modern" topics, such as network coding and coding for network coding, quantum error correction
- Lots of applications: Hashing, Shamir secret sharing, group testing, streaming and data structures, pseudorandom generators, and probablistically checkable proofs.

Thanks to Madhu and Chi-Ning for a great semester!