6.895 Essential Coding Theory October 13, 2004

Lecture 10

Lecturer: Madhu Sudan *Scribe: Jesse Kamp*

1 Overview

- Binary Codes
  - Concatenation
  - Algorithmic Issues

2 Recap

So far, we have studied 3 classes of codes:

1. Hamming Codes
   - $d = 3$, $n \to \infty$, $\frac{d}{n} \to \infty$ (as $n \to \infty$).
2. Reed-Solomon Codes
   - $[n, k, n-k+1]_n$ codes.
   - Alphabet size $q = n \to \infty$ as $n \to \infty$.
3. Reed-Muller
   - $[n, \frac{n}{(2m)^m}, \frac{n}{2}]_{n^{\frac{1}{m}}}$ codes (roughly).
   - Either $q \to \infty$ or $\frac{k}{n} \to \infty$ as $n \to \infty$.
   - Hadamard Codes (special case of Reed-Muller)
     - $[n, \log n, \frac{n}{2}]_2$ codes.
     - $\frac{k}{n} \to 0$ as $n \to \infty$.

Big Question: Can we get $q = 2$ and $\frac{k}{n}, \frac{d}{n} > 0$ explicitly?

3 Reed-Solomon Codes on CDs

We now show how Reed-Solomon codes are used to encode the data on CDs. First, the alphabet size is chosen to be $q = n = 2^t$ for some integer t. We start out with a $[n, \frac{n}{2}, \frac{n}{2}]_n$ code. But now, since n is a power of 2, we can write every element of $\mathbb{F}_n$ as a t-bit vector over $\mathbb{F}_2$ (using $\mathbb{F}_{2^t} \equiv \mathbb{F}_2^t$). This transforms the original code into a $[nt, \frac{nt}{2}, \frac{n}{2}]_2$ code. The distance comes from the fact that for each field element which differs in the original code, we must have at least one location differ in the corresponding t-bit vectors. Since $t = \log n$, this code has parameters $[n \log n, \frac{n \log n}{2}, \frac{n}{2}]_2$. Note that in transforming the RS code into a binary code, we lose a lot in the relative distance.

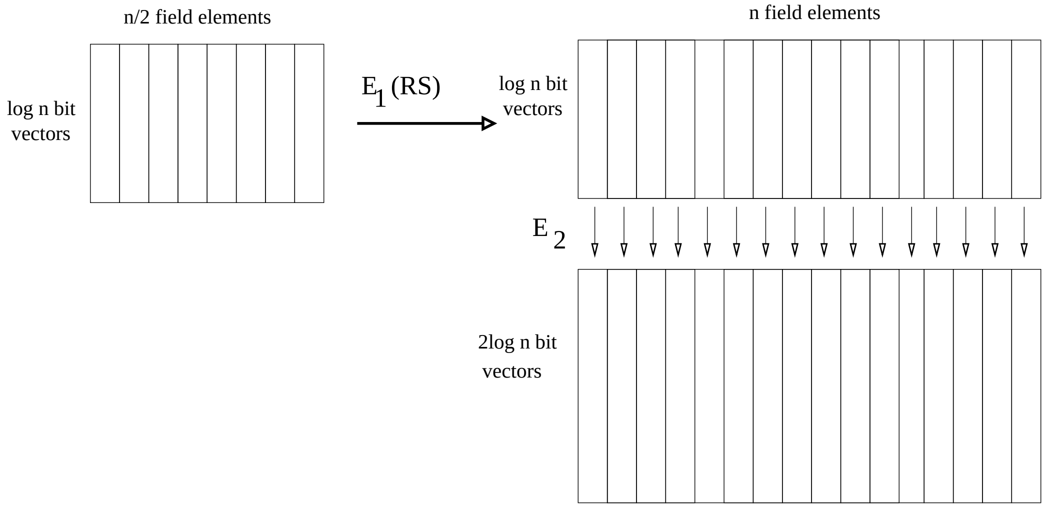
Now, instead of looking at distance $n/2$, we see what happens when we perform the same procedure with constant distance d (think of $d = 5$). We start out with a $[n, n-d+1, d]_n$ RS code and we end up with a $[n \log n, n \log n - d \log n + \log n, d]_2$ code. Letting $N = n \log n$ and thinking of $\log N$ $\log n$, this code is $[N, N - (d-1) \log N, d]_2$.

For $d = 3$, we can compare this with the Hamming code, which is $[N, N - \log N, 3]_2$. In comparison, we get $[N, N - 2 \log N, 3]_2$, so we're only off by the factor of 2 in k.

We can also ask what we should be losing asymptotically. The best known codes for constant d are the BCH codes, which are $[N, N - \frac{d-1}{2}\log N, d]_2$ codes. As in the Hamming case, we're only off by a factor of 2 in k.

Putting this all together, we see that for constant d, we get pretty close to optimal using RS encoding even using the naive method of encoding each field element as a t-bit vector. However, for d a constant fraction of n, we didn't do nearly as well. This loss comes directly from this naive method of encoding field elements. To do better, we can use error correcting codes to encode the field elements. This leads to the idea of concatenated codes.

4 Concatenated Codes



The above picture shows how a concatenated code works. First, we encode with E_1 (RS, for example). Then viewing each of the field elements in the codeword as $\log n$ bit vectors, we can encode each of them separately with a second code E_2, which maps $\log n$ bits to $2\log n$ bits.

Now suppose E_1 is our $[n, \frac{n}{2}, \frac{n}{2}]_n$ RS code (as before) and that image$(E_2) = [2\log n, \log n, \frac{\log n}{10}]_2$, which is about what we'd get randomly. The distance in the concatenated code is the product of the distances, which is $\frac{n}{2}\frac{\log n}{10} = \frac{n \log n}{20}$. Thus the concatenated code is $[2n\log n, \frac{n\log n}{2}, \frac{n\log n}{20}]_2$. We lose $\frac{1}{2}$ the rate and distance over E_2, but we get the benefit of a much longer code.

We can search for E_2 efficiently. Since searching takes time exponential in the length of the code, which is $O(\log n)$, it can be done in polynomial time. This result was discovered by Forney in 1966.

Theorem 1 (Forney) *In polynomial time in n, we can find an "asymptotically good" code of length n, rate Rn, and distance δn, for constants $R, \delta > 0$.*

This is good, but the question is, can we be even more explicit than this? We would like to be able to give the generator matrix G implicitly by an algorithm to computer $G_{i,j}$ in time poly$(|i|, |j|) =$ poly$(\log n)$. However, finding the generator matrix for E_2 takes time poly(n).

5 Justesen Construction

Forney's insight was that there exists a space of codes $C_1, ..., C_{\text{poly}(n)}$ such that for some i, C_i is a $[2\log n, \log n, \frac{\log n}{10}]_2$ code.

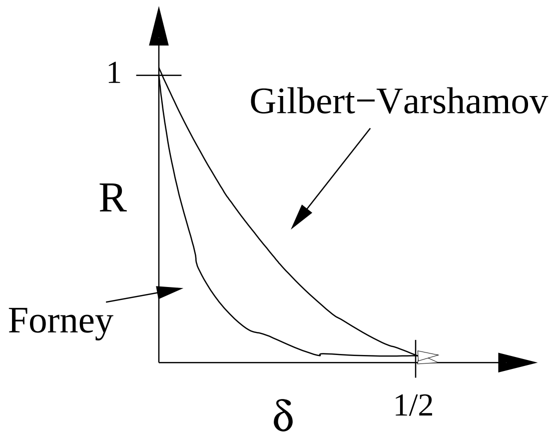


Figure 1: The above figure shows how the Forney codes compare to the Gilbert-Varshamov bound.

Justesen later improved on this and showed that there exists a space of codes $C_1, ..., C_n$ such that for 90% of all i's (roughly), C_i is a $[2\log n, \log n, \frac{\log n}{10}]_2$ code. (Note that it's not hard to get size exactly n here.)

Now note that there's no reason that E_2 needs to be the same everywhere. So we can apply C_i from the Justesen construction to the ith vector. Now 10% of the columns have bad C_i, but we're no worse of than if we'd started without these columns in the first place, in which case we'd still have a $[n, \frac{n}{2}, .4n]_n$ code. Thus concatenating our $[n, \frac{n}{2}, \frac{n}{2}]_n$ RS code with $(C_1, ..., C_n)$ gives us a $[2n\log n, \frac{n\log n}{2}, \frac{.4n\log n}{10}]_2$ code.

This gives us an improvement since unlike in the Forney case, we don't have to search for a good C_i. Thus this is more explicit, and the generator matrix can be implicitly computed. (Note that it's not immediately clear how to compute in poly$(\log n)$ time, but it certainly can be done in logspace.)

6 Justesen Ensemble $\Leftarrow$ Wozencraft

The construction of Justesen uses a code earlier given by Wozencraft. We index each of the codes by some $\alpha \in \mathbb{F}_q$, so we have $\{C_\alpha\}_{\alpha\in\mathbb{F}_q}$. (Here $\mathbb{F}_{q=2^t} \equiv \mathbb{F}_2^t$.)

The encoding process is as follows:

$$C_\alpha : x \in \mathbb{F}_2^t \to x' \in \mathbb{F}_{2^t} \to (x', \alpha \cdot x') \in \mathbb{F}_{2^t}^2 \to (x, \alpha(x)) \in \mathbb{F}_2^{2t}$$

It can be shown that for "most" α, C_α is good. Combining $\{C_\alpha\}_{\alpha\in\mathbb{F}_q}$ with the $[q, \frac{q}{2}, \frac{q}{2}]_q$ RS "outer" code, we get the Justesen bound $[2q\log q, \frac{q\log q}{2}, ...]_2$. Figure 2 illustrates the final encoding process.

Now note that all we've really done is use two polynomials instead of just one. Though this seems like a simple variation on the RS code, we don't know how to reason about this directly without going through the concatenation argument.

7 Decoding Concatenated Codes

The basic idea for decoding is to do a brute force search to find the nearest codeword in the inner code. This allows us to decode the inner code in time poly(n). Once we've done this, we can then decode the outer (RS) code.

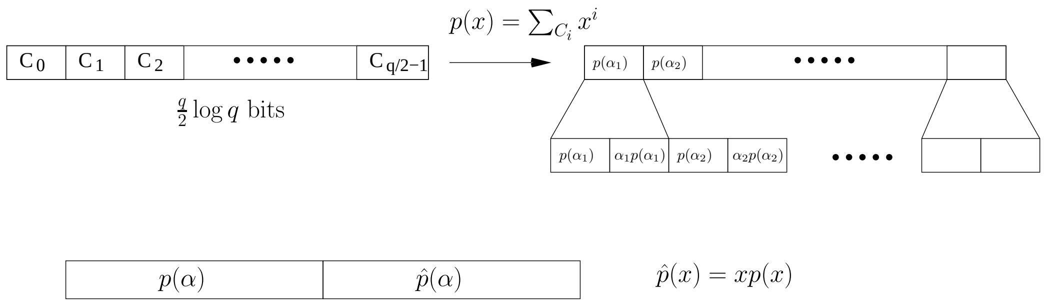


Figure 2: Justesen encoding process

7.1 Forney's Decoding

- $[n, k, d]_q$ - outer code: RS $\rightarrow$ can correct $\frac{d}{2}$ errors.
- $[n', k', d']_2$ - inner code (with $2^{k'} = q$) $\rightarrow$ can correct $\frac{d'}{2}$ errors in time poly$(2^{k'})$.
- $[nn', kk', dd']_2$ - concatenated code $\rightarrow$ can correct $\left(\frac{d}{2}\right)\left(\frac{d'}{2}\right) = \frac{dd'}{4}$ errors in "poly" time.
- Would like to be correct $\frac{dd'}{2}$ errors.

Forney's idea was to first observe that we can decode RS codes with e errors and s erasures provided $e < \frac{d-s}{2}$. When decoding the inner code, we can tell how many errors have happened (since there must have been at least as many errors as the distance to the nearest codeword, which we find by search). Let d_i be the distance between the ith vector in the recovered word and the closest codeword. Then let $e_i = \min\{\frac{d'}{2}, d_i\}$. Then instead of outputting the actual decoded vector v_i always, output v_i with probability $1 - \frac{e_i}{d'/2}$ and output an erasure with probability $\frac{e_i}{d'/2}$. The idea is that this forces the adversary to throw more errors into each vector. For example, if the adversary just put the minimum $\frac{d'}{2}$ errors into each vector, then we'd always note this as an erasure instead of an error and thus be better able to correct from it.