15-853: Algorithms in the Real World

Error Correcting Codes I
- Overview, Hamming Codes, Linear Codes

Error Correcting Codes II
- Reed-Solomon Codes
- Concatenated Codes

Error Correcting Codes III (LDPC/Expander Codes)
Error Correcting Codes IV (Decoding RS, Number Thy)

Block Codes
Each message and codeword is of fixed size
\[ \Sigma = \text{codeword alphabet} \]
\[ k = |m| \quad n = |c| \quad q = |\Sigma| \]
\[ C \subseteq \Sigma^n \text{ (codewords)} \]
\[ \Delta(x, y) \text{ = number of positions \ s.t. } x_i \neq y_i \]
\[ d = \min\{\Delta(x, y) : x, y \in C, x \neq y\} \]
Code described as: \((n, k, d)_q\)

Linear Codes
If \(\Sigma\) is a field, then \(\Sigma^n\) is a vector space

**Definition**: \(C\) is a linear code if it is a linear subspace of \(\Sigma^n\) of dimension \(k\).

This means that there is a set of \(k\) independent vectors \(v_i \in \Sigma^n\) (\(1 \leq i \leq k\)) that span the subspace.

i.e. every codeword can be written as:
\[ c = a_1 v_1 + a_2 v_2 + \ldots + a_k v_k \quad \text{where } a_i \in \Sigma \]

“Linear”: the sum of two codewords is a codeword.
Minimum distance = weight of least-weight codeword

Generator and Parity Check Matrices

**Generator Matrix**: A \(k \times n\) matrix \(G\) such that: \(C = \{xG \mid x \in \Sigma^k\}\)
Made from stacking the spanning vectors

**Parity Check Matrix**: An \((n - k) \times n\) matrix \(H\) such that: \(C = \{y \in \Sigma^n \mid Hy^T = 0\}\)
(Codewords are the null space of \(H\)).

These always exist for linear codes
Relationship of G and H

For linear codes, if G is in standard form \([I_k A]\) then \(H = [-A^T I_{n-k}]\).

Example of (7,4,3) Hamming code:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

Two Codes

<table>
<thead>
<tr>
<th>Hamming codes are binary ((2^r-1-l, 2^r-1-r, 3)) codes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basically ((n, n - \log n, 3))</td>
</tr>
<tr>
<td>Hadamard codes are binary ((2^r-1, r, 2^r-1)).</td>
</tr>
<tr>
<td>Basically ((n, \log n, n/2))</td>
</tr>
</tbody>
</table>

The first has great rate, small distance.
The second has poor rate, great distance.
Can we get \(\Omega(n)\) rate, \(\Omega(n)\) distance?

Yes. One way is to use a random linear code.
Let’s see some direct, intuitive ways.

Reed-Solomon Codes

Irving S. Reed and Gustave Solomon
Reed-Solomon Codes in the Real World

- **(204,188,17)\textsubscript{256}**: ITU J.83(A)\textsuperscript{2}
- **(128,122,7)\textsubscript{256}**: ITU J.83(B)
- **(255,223,33)\textsubscript{256}**: Common in Practice
  - Note that they are all byte based (i.e., symbols are from GF(2^8)).

Decoding rate on 1.8GHz Pentium 4:
- (255,251) = 89Mbps
- (255,223) = 18Mbps

Dozens of companies sell hardware cores that operate 10x faster (or more)
- (204,188) = 320Mbps (Altera decoder)

Applications of Reed-Solomon Codes

- **Storage**: CDs, DVDs, “hard drives”
- **Wireless**: Cell phones, wireless links
- **Satellite and Space**: TV, Mars rover, Voyager,
- **Digital Television**: DVD, MPEG2 layover
- **High Speed Modems**: ADSL, DSL, ..

Good at handling burst errors.
Other codes are better for random errors.
  - e.g., Gallager codes, Turbo codes

Viewing Messages as Polynomials

A \((n, k, n-k+1)\) code:
Consider the polynomial of degree \(k-1\)
\[ p(x) = a_{k-1} x^{k-1} + \cdots + a_1 x + a_0 \]

**Message**: \((a_{k-1}, \ldots, a_1, a_0)\)

**Codeword**: \((p(1), p(2), \ldots, p(n))\)

To keep the \(p(i)\) fixed size, we use \(a_i \in GF(q^r)\)
To make the \(i\) distinct, \(n \leq q^r\)

For simplicity, imagine that \(n = q^r\). So we have a \((n, k, n-k+1)\) code.
Alphabet size increasing with the codeword length.
A little awkward. (But we can still do things.)
**Which field to use**

The general ideas will not depend on the field.
Assume we're working in \( \mathbb{F}_q \) where \( q \) is a prime.
Just has to be large enough to evaluate at \( n \) points.

In practice, use \( q = 2^7 \), e.g., \( \mathbb{F}_{2^8} = \mathbb{F}_{256} \).
Recall: this is not the same as the vector space \( (\mathbb{F}_2)^8 \).
Always think about \( \mathbb{F}_{2^8} \) in terms of polynomials.

---

**Polynomial-Based Code**

A \((n, k, 2s + 1)\) code:

\[
\begin{array}{c}
\text{k} \\
\hline
\text{n} \\
\end{array}
\begin{array}{c}
\text{2s} \\
\end{array}
\]

Can **detect** \( 2s \) errors
Can **correct** \( s \) errors
Generally can correct \( \alpha \) erasures and \( \beta \) errors if \( \alpha + 2\beta \leq 2s \)

---

**Polynomials and their degrees**

**Fundamental theorem of Algebra:**
Any non-zero polynomial of degree \( k \) has at most \( k \) roots (over any field).

**Corollary 1:**
If two degree-\( k \) polynomials \( P, Q \) agree on \( k+1 \) locations (i.e., if \( P(x_i) = Q(x_i) \) for \( x_0, x_1, ..., x_k \)), then \( P = Q \).

**Corollary 2:**
Given any \( k+1 \) points \((x_i, y_i)\), there is at most one degree-\( k \) polynomial that has \( P(x_i) = y_i \) for all these \( i \).

---

**Theorem:**
Given any \( k+1 \) points \((x_i, y_i)\), there is exactly one degree-\( k \) polynomial that has \( P(x_i) = y_i \) for all these \( i \).

**Proof:** e.g., use Lagrange interpolation.
Why is the distance \( n-k+1 \)?

**Direct Proof**

1. RS is a linear code: indeed, if we add two codewords corresponding to \( P(x) \) and \( Q(x) \), we get a codeword corresponding to the polynomial \( P(x) + Q(x) \).

2. So look at the least weight codeword. It is the evaluation of a polynomial of degree \( k-1 \) at some \( n \) points. So it can be zero on only \( k-1 \) points. Hence non-zero on \( (n-(k-1)) \) points.

3. This means distance at least \( d = n-k+1 \)

---

**Correcting Errors**

**Correcting \( s \) errors:**

1. Find \( k+s \) symbols that agree on a degree \( (k-1) \) poly \( p(x) \). These must exist since originally \( k+2s \) symbols agreed and only \( s \) are in error

2. There are no \( k+s \) symbols that agree on the wrong degree \( (k-1) \) polynomial \( p'(x) \)
   - Any subset of \( k \) symbols will define \( p'(x) \)
   - Since at most \( s \) out of the \( k+s \) symbols are in error, \( p'(x) = p(x) \)

   This suggests a brute-force approach, very inefficient. Better algos exist (talk on the board).

---

**The Berlekamp-Welch Decoder**

An efficient way to decode Reed-Solomon codes.

We may do this in coding lecture #4 this time.

---

**RS and “burst” errors**

Let’s compare to Hamming Codes (which are also optimal).

<table>
<thead>
<tr>
<th>Code</th>
<th>code bits</th>
<th>check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS ((255, 253, 3)_{256})</td>
<td>2040</td>
<td>16</td>
</tr>
<tr>
<td>Hamming ((2^{11\cdot-1}, 2^{11\cdot1-1}, 3)_{2})</td>
<td>2047</td>
<td>11</td>
</tr>
</tbody>
</table>

They can both correct 1 error, but not 2 random errors.

- The Hamming code does this with fewer check bits
  However, RS can fix 8 contiguous bit errors in one byte
  - Much better than lower bound for 8 arbitrary errors
    
    \[
    \log \left(1 + \binom{n}{1} + \cdots + \binom{n}{8}\right) > 8\log(n-7) \approx 88 \text{ check bits}
    \]
One way to achieve linear rate and linear distance

**CONCATENATED CODES**

---

**Concatenated Codes**

Take a RS code \((n, k, n-k+1)_q = n\) code.

Can encode each alphabet symbol of \(k' = \log q = \log n\) bits using another code.

E.g., use \(((k' + \log k'), k', 3)\)_2-Hamming code. Now we can correct one error per alphabet symbol with little rate loss. (Good for sparse periodic errors.)

Or \((2^k, k', 2^{k-1})_2\) Hadamard code. (Say \(k = n/2\).) Then get \((n^2, (n/2) \log n, n^2/4)_2\) code.

Much better than plain Hadamard code in rate, distance worse only by factor of 2.
Concatenated Codes

Take a RS code \((n,k,n-k+1)_{q,n}\) code. Can encode each alphabet symbol of \(k' = \log q = \log n\) bits using another code.

Or, since \(k'\) is \(O(\log n)\), could choose a code that requires exhaustive search but is good. Random linear codes give \(((1+f(\delta))k', k', \delta k')_q\) codes. Composing with RS (with \(k = n/2\)), we get 
\(( (1+f(\delta))n \log n, (n/2) \log n, \delta(n/2)\log n )_2\)

Gives constant rate and constant distance and binary alphabet! And poly-time encoding and decoding.