

Lecture 14 : 03.04.04

*Lecturer: Satish**Scribe: Russell Sears*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

14.1 Introduction to Erasure Coding

The idea behind erasure coding is to split a file f into n pieces so that the original file can be reproduced using any $m < n$ pieces of the file. For instance, you might stripe the file across n disks, and still be able to recover the data if one disk crashed. We want to do this without using “too much” space.

14.1.1 Lower bound on space

If we break the file into n pieces, and are able to recover from $n - 1$ of those pieces, then those $n - 1$ pieces must take up at least as much space as the original file. Otherwise, we would be able to recover more data than was transmitted, given an arbitrary message. Information theory says that this is not possible. Therefore, the n pieces of the encoded file must take up at least $(1 + \frac{1}{n-1}) * |f|$ space.

14.1.2 Parity checking

Parity bits can be used to implement a simple erasure code. (In fact, it is the scheme used in RAID 5.) The data is split evenly across the first $n-1$ chunks, and the n 'th chunk contains the XOR of the other chunks. It is straightforward to recover the contents when one chunk has been lost.

However, this scheme only works when you only are interested in recovering from a single failure, and when you know which chunk has been lost or corrupted. The rest of this lecture will address these two problems. Another concern is that these algorithms assume that corruption occurs at the block level. We would rather deal with corruption at the bit level.

14.2 Reed-Solomon Encoding

Given k linearly independent equations of k unknowns, one can solve for the k unknowns. The Reed-Solomon code transforms a message into a polynomial, evaluates the polynomial at many points, and then transmits the results of these evaluations.

The sender splits the message into $n > k$ pieces, where k is high enough to allow the message to be represented by the coefficients of the polynomial, and $n - k$ is the number of erasures that can be tolerated.

Upon receiving any k of these evaluations, the recipient can solve for the original polynomial, and recover the message.

The polynomial looks like this:

$$f_{k-1}x^{k-1} + f_{k-2}x^{k-2} + \dots + f_0$$

where the set of f_i coefficients are a numerical representation of the original data.

The values of x that the polynomial is evaluated at can be decided ahead of time so that they do not have to be transmitted. Therefore, the size of the data that has to be transmitted is equal to the size of n coefficients.

We can think of the encoding process in terms of the following linear algebra equation, where the first matrix has n rows and k columns. We need each f_m to contain $\frac{m}{k}$ bits so that we have enough room to store the original message. To encode, we simply evaluate the left hand side. Y contains the values that should be sent.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{k-1} \end{bmatrix} \begin{bmatrix} f_0 \\ \vdots \\ f_{k-1} \end{bmatrix} = Y$$

We've described the encoding process, but we still need to come up with a way of reconstructing the data, and we haven't talked about the space requirements for the polynomials' evaluations.

14.2.1 Decoding

To decode, we can multiply Y by the inverse of the first matrix above, or do Gaussian elimination. Since the above system is over-constrained, we can simply delete corresponding rows from the first matrix and Y until there are only k remaining rows. As long as we receive at least k pieces of the message, we can delete rows corresponding to the lost pieces and reconstruct the original message.

Normally, we think of operations on polynomials in terms of the real numbers. Unfortunately, floating point calculations are subject to precision errors, which makes it difficult to reason about the size of field elements, or the precision required for operations during encoding and decoding. Instead, we will reformulate our algorithm to use discrete operations.

14.2.2 Galois Fields

Definition 14.1 *Field, $\langle A(+, 0)(*, 1) \rangle$*

A field is a non-empty set A with operations $+$ and $$, where both operations are commutative and associative, and have additive and multiplicative inverses for all elements of the field (except zero, which has no multiplicative inverse), and have additive and multiplicative identities 0 and 1 , respectively.*

Note that it follows from this definition that the $+, -, *, /$ operations are guaranteed to have a value for all pairs of field members. Since these are the only operations needed to perform operations like matrix multiplication, inversion and Gaussian elimination, all of the operations that we performed on the above polynomials will hold under any field.

For example, rational numbers form a field. Standard addition is denoted with the $+$ symbol, and has identity 0 and the inverse of n is $-n$. Multiplication is denoted with the $*$ symbol, has identity 1 , and has inverse of $n = \frac{1}{n}$.

Note that the integers are not a field (since some integers do not have integer multiplicative inverses), for example:

$$\begin{aligned} x + y &= 4 \\ x - y &= 7 \end{aligned} \Rightarrow x = \frac{11}{2}, y = -1\frac{1}{2}$$

Instead of using rational or real numbers, we use “Galois Fields” defined by modular arithmetic over primes:

$$GF(p) = \langle Z_p(+_p, 0)(*_p, 1) \rangle$$

Multiplication and addition are simply mod p . For example:

$$\begin{aligned} x + y &= 4 \\ x - y &= 7 \end{aligned} \pmod{7} \Rightarrow x = 2, y = 2$$

Proof: For all p prime, $GF(p)$ is a field.

Clearly, the identities are simply 0 and 1. Furthermore, it is relatively easy to see that the additive inverses exist.

Multiplicative inverses are a bit trickier, but can be proven using the fact that we chose a prime value for p . Intuitively, we need to show that each number has a multiplicative inverse, (For example, 2’s multiplicative inverse is 4, and 4’s is 2.) and that when multiplied together, they equal 1. Since we cannot be ‘in step’ with the prime, we must eventually ‘hit’ every number. (Otherwise, p and a number in the field must have some common divisor, but p is prime.)

The extended GCD algorithm proves that if $GCD(x, y) = 1$ then $\exists a, b$ s.t. $ax + by = 1$. If we substitute p for y , and mod both sides of the equation by p , we obtain $ax + 0 = 1$, which implies that a is the multiplicative inverse of x . Since we can perform these operations with the operators provided by the field, both a and x are members of the field.

■

14.2.3 Space utilization

We can pick any prime number that we want, and get a field that it is “big enough.” In particular, if m is the length of the message in bits, and k is the number of coefficients, we need to specify m/k bits with each coefficient in the polynomial, with the field, so we need to pick $p > 2^{m/k}$. Also, we need to choose $p > n$ if we are going to evaluate the polynomial at n points.

Just how good is the space utilization of this algorithm? Ignoring the difference between p and $2^{m/k}$, it’s perfect. Any combination of k blocks, use m bits of storage, and allow us to decode our m bit message.

Reconstructing the original file is a bit slow, since it requires quadratic work. There are variations on the algorithm that decode quickly, but require $k + \epsilon$ blocks to reconstruct.

14.3 Error Correction

Say there are e errors, which cause block y to be replaced with some other data, y' :

$$y_0 \dots y_n \rightarrow y_0 \dots y'_i \dots y'_j \dots y_n$$

If we knew which blocks contained errors, we could simply discard them and reconstruct the message using erasure codes.

Unfortunately, in practice, we don't always know which ones contain errors. Recall that a message can be reconstructed provided that the number of erasures is less than or equal to $n - k$. If we consider errors, it turns out that that a message can be reconstructed when the number of errors, $e \leq \frac{n-k-1}{2}$, which will be argued later. In other words, you can only correct half as many errors as you can tolerate erasures.

14.3.1 Berlekamp-Welsh

Consider the following "error polynomial":

$$E(x_i) = \{0 \text{ if } y'_i \neq y_i; \neq 0 \text{ if } y'_i = y_i\}$$

If we had this polynomial, then we'd be done. Of course, the sender cannot compute it, since it requires knowledge of all of the y'_i .

Note that the following polynomial has the form that we need for E:

$$E(x) = \prod_{y'_i \neq y_i} (x - y'_i)$$

Note that it has terms: $x^e + E_{e-1}x^{e-1} \dots E_0$.

So, we have two polynomials:

$$\begin{aligned} f(x) &= f_{k-1}x^{k-1} \dots f_0 \\ E(x) &= x^e + E_{e-1}x^{e-1} + \dots \end{aligned}$$

Multiplying them, we get:

$$g(x) = E(x)f(x) = g_{k+e-1}x^{k+e-1} \dots g_0$$

Without errors, the recipient would get: $f(x_1) = y_1, f(x_2) = y_2, \dots$

Since there can be errors in the recipient receives y' instead of y , but note that:

$$\begin{aligned} E(x_1)f(x_1) &= y'_1E(x_1) \\ &\dots \\ E(x_n)f(x_n) &= y'_nE(x_n) \end{aligned}$$

So, in the rows where there is an error, then $E \neq$ zero, and we can solve for the y'_1 . If there is no error, then both sides are equal to zero, so that works too.

Now, we don't know E, but we do know its form. Note that the left hand side is just g, so we have:

$$g(x_1) = y'_1E(x_1)$$

which multiplies out to:

$$g_{k+e-1}x_1^n \dots g_0 = y'_1x_1^n + y'_1E_0,$$

However, we know the y_n, x_n , and the rest is just a linear equation in $k+e-1$ unknowns, and we have n equations on these unknowns, so we can solve for E and g as long as $\#equations \geq \#unknowns, n \geq k+2e-1$. This lets us find g and e, but we want to find f:

$$f(x_i) = \frac{g(x_i)}{E(x_i)}$$

Solving for n shows us that $e \leq \frac{n-k+1}{2}$

(We left out the proof that the equations are all linearly independent in the above analysis.)

14.4 Big picture

Error correcting and erasure codes 'smooth out' information from the original data across many smaller pieces.

There are still some problems with the algorithms that we've presented. We've been talking about corruption of single characters, but in practice we are more interested in bit streams. For example, if 50% of the characters are corrupted, that might be tolerable, but if 50% of the bits are incorrect, then almost none of the characters will be intact.