Reed-Solomon Codes

Reed-Solomon codes

In these notes we examine Reed-Solomon codes, from a computer science point of view.

Reed-Solomon codes can be used as both error-correcting and erasure codes. In the error-correcting setting, we wish to transmit a sequence of numbers over a noisy communication channel. The channel noise might cause the data sent to arrive corrupted. In the erasure setting, the channel might fail to send our message. For both cases, we handle the problem of noise by sending additional information beyond the original message. The data sent is an *encoding* of the original message. If the noise is small enough, the additional information will allow the original message to be recovered, through a *decoding* process.

Encoding

Let us suppose that we wish to transmit a sequence of numbers $b_0, b_1, \ldots, b_{d-1}$. To simplify things, we will assume that these numbers are in $GF(p)$, i.e., our arithmetic is all done modulo p. In practice, we want to reduce everything to bits, bytes, and words, so we will later discuss how to compute over fields more conducive to this setting, namely fields of the form $GF(2^r)$.

Our encoding will be a longer sequence of numbers $e_0, e_1, \ldots, e_{n-1}$, where we require that $p > n$. We derive the e sequence from our original b sequence by using the b sequence to define a polynomial P, which we evaluate at n points. There are several ways to do this; here are two straightforward ones:

- Let $P(x) = b_0 + b_1 x + b_2 x^2 + \ldots b_{d-1} x^{d-1}$. This representation is convenient since it requires no computation to define the polynomial. Our encoding would consist of the values $P(0), P(1), \ldots, P(n-1)$. (Actually, our encoding could be the evalution of P at any set of n points; we choose this set for convenience. Notice that we need $p > n$, or we can't choose n distinct points!)
- Let $P(x) = c_0 + c_1 x + c_2 x^2 + \ldots c_{d-1} x^{d-1}$ be such that $P(0) = b_0$, $P(1) = b_1$, $\ldots$, $P(d-1) = b_{d-1}$. Our encoding $e_0, e_1, \ldots, e_{n-1}$ would consist of the values $P(0), P(1), \ldots, P(n-1)$. Although this representation requires computing the appropriate polynomial P, it has the advantage that our original message is actually sent as part of the encoding. A code with this property is called a *systematic* code. Systematic codes can be useful; for example, if there happen to be no erasures or errors, we might immediately be able to get our message on the other end.

  The polynomial $P(x)$ can be found by using a technique known as Lagrange interpolation. We know that there is a unique polynomial of degree $d-1$ that passes through d points. (For example, two points uniquely determine a line.) Given d points $(a_0, b_0), \ldots, (a_{d-1}, b_{d-1})$, it is easy to check that

  $$P(x) = \sum_{j=0}^{d-1} b_j \prod_{k \neq j} \frac{x - a_k}{a_j - a_k}$$

  is a polynomial of degree $d-1$ that passes through the points. (To check this, note that $\prod_{k \neq j} \frac{x - a_k}{a_j - a_k}$ is 1 when $x = a_j$ and 0 when $x = a_k$ for $k \neq j$.)

Note that in either case, the encoded message is just a set of values obtained from a polynomial. The important point is not which actual polynomial we use (as long as the sender and receiver agree!), but just that we use a polynomial that is uniquely determined by the data values.

We will also have to assume that the sender and receiver agree on a system so that when the encoded information e_i arrives at the receiver, the receiver knows it corresponds to the value $P(i)$. If all the information is sent and arrives in a fixed order, this of course is not a problem. In the case of erasures, we must assume that when an encoded value is missing, we know that it is missing. For example, when information is sent over a network, usually the value i is derived from a packet header; hence we know when we receive a value e_i which number i it corresponds to.

Decoding

Let us now consider what must be done at the end of the receiver. The receiver must determine the polynomial from the received values; once the polynomial is determined, the receiver can determine the original message values. (In the first approach above, the coefficients of the polynomial are the message values; in the second, given the polynomial the message is determined by computing $P(0)$, $P(1)$, etc.)

Let us first consider the easier case, where there are erasures, but no errors. Suppose that just d (correct) values $e_{j_1}, e_{j_2}, \ldots, e_{j_d}$ arrive at the receiver. No matter which d values, the receiver can determine the polynomial, just by using Lagrange interpolation! Note that the polynomial the receiver computes must match P, since there is a unique polynomial of degree $d-1$ passing through d points.

What if there are errors, instead of erasures? (By the way, if there are erasures and errors, notice that we can pretend an erasure is an error, just by filling the erased value with a random value!) This is much harder. When there were only erasures, the receiver knows which values they received and that they are all correct. Here the receiver has obtained n values, $f_0, f_1, \ldots, f_{n-1}$, but has no idea which ones are correct.

We show, however, that as long as at most k received values are in error, that is $f_j \neq e_j$ at most k times, then the original message can be determined whenever $k \leq \frac{n-d}{2}$. Notice that we can make n as large as we like. If we know a bound on error rate in advance, we can choose n accordingly. By making n sufficiently large, we can deal with error rates of up to 50%. (Of course, recall that we need $p > n$.)

The decoding algorithm we cover is due to Berlekamp and Welch. An important concept for the decoding is an *error polynomial*. An error polynomial $E(x)$ satifies $E(i) = 0$ if $f_i \neq e_i$. That is, the polynomial E marks the positions where we have received erroneous values. Without loss of generality, we can assume that E has degree k and is *monic* (that is, the leasing coefficient is 1), since we can choose E to be$\prod_{i: f_i \neq e_i}(x-i)$ if there are k errors, and throw extra terms $x-i$ in the product where f_i equals e_i if there are fewer than k errors.

Now consider the following interesting (and somewhat magical) setup: we claim that there exist polynomials $V(x)$ of degree at most $d-1+k$ and monic $W(x)$ of degree at most k satisfying

$$V(i) = f_i W(i).$$

Namely, we can let $W(x) = E(x)$, and $V(x) = P(x) \cdot E(x)$. It is easy to check that $V(i)$ and $W(i)$ are both 0 when $f_i \neq e_i$, and are both $e_i W(i)$ otherwise. So, if someone handed us these polynomials V and W, we could find $P(x)$, since $P(x) = V(x)/W(x)$.

There are two things left to check. First, we need to show how to find polynomials V and W satisfying $V(i) = f_i W(i)$. Second, we need to check that when we find these polynomials, we don't somehow find a wrong pair of polynomials that do not satisfy $V(x)/W(x) = P(x)$. For example, a priori we could find a polynomial D that was different from E!

First, we show that we can find an V and W efficiently. Let $v_0, v_1, \ldots, v_{d+k-1}$ be the coefficients of V and $w_0, w_1, \ldots, w_k$ be the coefficients of W. Note we can assume $w_k = 1$. Then the equations $V(i) = f_i W(i)$ give n linear equations in the coefficients of V and W, so that we have n equations and $d + 2k \leq n$ unknowns. Hence a pair V and W can be found by solving a set of linear equations.

Since we know a solution V and W exist, the set of linear equations will have a solution. But it could also have many solutions. However, any solution we obtain will satisfy $V(x)/W(x) = P(x)$. To see this, let us suppose we have two pairs of solutions (V_1, W_1) and (V_2, W_2). Clearly $V_1(i)W_2(i) f_i = V_2(i)W_1(i) f_i$. If f_i does not equal 0, then $V_1(i)W_2(i) = V_2(i)W_1(i)$ by cancellation. But if f_i does equal 0, then $V_1(i)W_2(i) = V_2(i)W_1(i)$ since then $V_1(i) = V_2(i)$ = 0. But this means that the polynomials V_1W_2 and V_2W_1 must be equal, since they agree on n points and each has degree $d+2k-1 < n$. But if these polynomials are equal, then $V_1(x)/W_1(x) = V_2(x)/W_2(x)$. Since any solution (V,W) yields the same ratio $V(x)/W(x)$, this ratio must always equal $P(x)$!

Arithmetic in $GF(2^r)$

In practice, we want our Reed-Solomon codes to be very efficient. In this regard, working in $GF(p)$ for some prime is inconvenient, for several reasons. Let us suppose it is most convenient if we work in blocks of 8 bits. If we work in $GF(251)$, we are not using all the possibilities for our eight bits. Besides being wasteful, this is problematic if our data (which may come from text, compressed data, etc.) contains a block of eight bits which corresponds to the number 252!

It is therefore more natural to work in a field with 2^r elements, or $GF(2^r)$. Arithmetic is this field is done by finding an irreducible (prime) polynomial $\pi(x)$ of degree r, and doing all arithmetic in $Z_2[\pi(x)]$. That is, all coefficients are modulo 2, arithmetic is done modulo $\pi(x)$, and $\pi(x)$ should not be able to be factored over $GF(2)$.

For example, for $GF(2^8)$, an irreducible polynomial is $\pi(x) = x^8 + x^6 + x^5 + x + 1$. A byte can naturally be though of as a polynomial in the field. For example, by letting the least significant bit represent x^0, and the ith least significant bit represent x^i, we have that the byte 10010010 represents the polynomial $x^7 + x^4 + x$. Adding in $GF(2^r)$ is easy: since all coefficients are modulo 2, we can just XOR two bytes together. For example

$$\begin{aligned} 10010010 + 10101010 &= 00111000 \\ (x^7 + x^4 + x) + (x^7 + x^5 + x^3 + x) &= x^5 + x^4 + x^3. \end{aligned}$$

Moreover, subtracting is just the same as adding!

Multiplication is slightly harder, since we work modulo $\pi(x)$. As an example

$$(x^4 + x) \cdot (x^4 + x^2) = x^8 + x^6 + x^5 + x^3.$$

However, we must reduce this so that we can fit it into a byte. As we work modulo $\pi(x)$, we have that $\pi(x) = 0$, or $x^8 = x^6 + x^5 + x + 1$. Hence

$$(x^4 + x) \cdot (x^4 + x^2) = x^8 + x^6 + x^5 + x^3 = (x^6 + x^5 + x + 1) + x^6 + x^5 + x^3 = x^3 + x + 1,$$

and hence

$$00010010 \cdot 00010100 = 00001011.$$

Rather than compute these products on the fly, all possible $256 \cdot 256$ pairs can be precomputed once in the beginning, and then all multiplications are done by just doing a lookup in the multiplication lookup table. Hence by using memory and preprocessing, one can work in $GF(2^8)$ and still obtain great speed.

Reed-Solomon codes work exactly the same over $GF(2^r)$ as they do over $GF(p)$, since in both cases the main reqiurement, namely that a polynomial of degree $d-1$ be uniquely defined by d points, is satisfied.