

Lectures 5 and 6 – A theorist’s toolkit

“Polynomials are fantastic”

1 Overview

We will start out with a discussion of error-correcting codes, and specifically look at Reed-Solomon codes and how they work. We will see how many errors they can correct and how errors are corrected. We will also briefly discuss how we can implement this on binary computers by looking at how a code in $\text{GF}[2^\ell]$ can be transformed into a code in $\text{GF}[2]$ while maintaining good error-correcting capabilities.

In the second part, we will look at Shamir’s secret sharing and its application in multi-party computations. With secret sharing, we can share a secret to n parties in such a way that t or more collaborating players can recover the secret, while $t - 1$ or fewer do not learn anything about the secret. With multi-party computations, we can compute the output of a function $f(X_1, \dots, X_n)$ where each player has one (secret) part of the input without leaking any unnecessary information about X_i .

2 Error-correcting codes

Perhaps a brief reminder of what an error-correcting code is, and what properties are important in such a code is in order. The purpose of an error-correcting code is to add some redundant information to a message in such a way that the message can be recovered, even if some part of the information is modified during transmission.

Let the message be a sequence of elements in $\text{GF}[q]$. We then say that a code has length n and dimension d if the code transforms a sequence of d elements into a message of size n . The density of a code is q^{d-n} . A code C is said to be *linear* if, whenever the codewords $a, b \in C$, then $a + b \in C$, where addition is componentwise in $\text{GF}[q]$.

We define a distance function $d(x, y)$ as the number of positions in which x and y differ. This distance is usually called the *Hamming distance*. For a code C , we are generally interested in the minimum distance, $\delta = \min_{x, y \in C, x \neq y} d(x, y)$. For linear codes, we have $\delta = \min_{x \in C, x \neq 0} d(x, 0)$. Note that if the number of errors is $< \delta/2$ we can, in principle, correct the error, since there will be a unique closest code word.

3 Reed-Solomon codes

In Reed-Solomon codes, a message looks like $(a_0, \dots, a_{n-k}) \in (\text{GF}[q])^{n-k+1}$, where $q \geq n$ and k is a parameter. We usually have $q \approx n$. Define the polynomial $P(x) = \sum_{i=0}^{n-k} a_i x^i$. This polynomial is evaluated in n different $\alpha_i \in \text{GF}[q]$ and the message to be transmitted is $(P(\alpha_1), \dots, P(\alpha_n))$. The α_i 's are assumed to be known by the sender and receiver. For instance, if we have a generator g for the non-zero group in $\text{GF}[q]$, we can let $\alpha_i = g^i$.

It is easy to see that this code has length n , dimension $n - k + 1$ and density $\frac{q^{n-k+1}}{q^n} = q^{1-k}$. But what is the minimum distance δ ? The code is linear, so

$$\delta = \min_{P(x)} \#(\alpha_i \mid P(\alpha_i) \neq 0) = \min_{P(x)} n - \#(\alpha_i \mid P(\alpha_i) = 0)$$

and since $P(x)$ is a polynomial of degree $n - k$, $\#(\alpha_i \mid P(\alpha_i) = 0) \leq n - k$ which gives $\delta \geq k$ and it is easy to see that we have equality. Thus, we should be able to correct errors in up to $t < \frac{k}{2}$ positions.

Let us compare Reed-Solomon codes to a random code with the same length and density, q^{1-k} and with minimal distance δ . If we assume that $q \approx n$ then we have, for small δ , at least

$$\sim \binom{n}{\delta} q^\delta \approx \frac{q^{2\delta}}{\delta!}$$

elements at distance at most δ from a fixed element. Hence we would expect the minimal distance to satisfy

$$\frac{q^{2\delta}}{\delta!} \approx q^{k-1}$$

giving $\delta \approx \frac{k-1}{2}$ but in Reed-Solomon codes we get $\delta = k$, i.e. we beat the random code by a factor 2.

4 Error correction in Reed-Solomon

Let us assume that $k = 2t + 1$ where t is the number of errors, and we have just received a message $(y_1, \dots, y_n)$. Thus, we know that $y_i = P(\alpha_i)$ for $n - t$ of the i 's. This means that there is a non-zero polynomial Q of degree $\leq t$ such that $Q(\alpha_i)y_i = R(\alpha_i)$ for all i , where R is a polynomial of degree $\leq n - k + t$ and, in fact, $R = QP$. To see that this is the case, let $Q(x)$ be a polynomial with $Q(\alpha_j) = 0$ for all $y_j \neq P(\alpha_j)$. There are at most t such points, so Q is a polynomial of degree $\leq t$.

A linear equation system can be set up to solve for the coefficients of Q and R , using $Q(\alpha_i)y_i = R(\alpha_i)$. We have n equations, and the total number of unknowns are $n - k + t$ coefficients in R and t coefficients in Q . Thus, a pair of polynomials, Q and R , can be found.

What we have now is thus two polynomials, Q and R , which we would like to use to calculate the message, the polynomial P . This can be done with straightforward polynomial division, $P = R/Q$. To see that this is the case, consider the equation

$$Q(\alpha_i)P(\alpha_i) = R(\alpha_i),$$

which we know is true for all correct y_i . The polynomials on both sides have degree $n - k + t = n - t - 1$. Since we know that the equation is true in $n - t$ points (i.e. more than the degrees), we conclude that $Q(x)P(x) = R(x)$ as polynomials, and thus calculating R/Q we indeed recover the correct polynomial P .

5 Codes in $\text{GF}[2^\ell]$

Assume we have a Reed-Solomon code with $n = q = 2^\ell$ and $k = n/2$. Then the length is n , the dimension is $n/2$ and the distance is $n/2$. If we consider this as a code over $\text{GF}[2]$ with the standard coding of $\text{GF}[2^\ell]$ (i.e. we look at bit-errors), we end up with a code with length $n\ell$, dimension $n\ell/2$ and distance $\geq n/2$. The minimal distance might be larger but it is hard to find a proof for this and we need to modify the construction to improve it in a provable way.

This can be done either by using a better coding of $\text{GF}[2^\ell]$ with twice the length of the standard code in an error correcting way. We would use an inner error correcting code with a minimal distance which is linear in ℓ . As ℓ is quite small this inner code can be found with exhaustive search. Another way to proceed is as follows. Transform element α in position j to $(\alpha, \alpha \cdot \beta_j)$ (again doubling the length) where $\beta_j \in \text{GF}[2^\ell]$ and $\beta_i \neq \beta_j, i \neq j$. With this

done, use any coding of $\text{GF}[2^\ell]$ by ℓ bits, e.g. the standard coding, works and it turns out that this gives a linear code of linear minimal distance and is in fact known as a Justesen code. We leave the details to prove that this indeed the case to the reader.

6 Shamir's secret sharing

In Shamir's secret sharing scheme a polynomial is used to share a secret with n players in such a way that t or more players must cooperate to recover the secret. Let the secret to be shared be $S = s_0$, the constant term of a polynomial

$$P(x) = \sum_{i=0}^{t-1} s_i x^i,$$

where the other coefficients are chosen randomly, uniformly and independently. We fix a set of values $\alpha_1, \dots, \alpha_n$ where $\alpha_i \neq \alpha_j, i \neq j$ and $\alpha_i \neq 0$. For the moment we assume that these are fixed but arbitrary public elements, but let us mention that sometimes it makes sense to have them random and/or known only to the player in question. Player i is given $P(\alpha_i)$ as his share of the secret.

As can be seen, $t - 1$ players cooperating have a polynomial of degree $t - 1$ evaluated in $t - 1$ points, and thus cannot recover the secret. This follows as there is a polynomial P_0 of degree $t - 1$ that equals 0 on the given point but $P_0(0) \neq 0$ and adding a multiples of P_0 to P shows that all secrets are possible and in fact equally likely. As soon as t players cooperate, however, the polynomial can be interpolated using standard techniques and the constant term recovered.

An evil player can pretend to cooperate and report a false value when everyone is pooling their secret, thus changing the secret. Error-correcting codes can be used to prevent this type of cheating as finding the encoding polynomial is equivalent to decoding a Reed-Solomon code.

Two questions were left as exercises for the audience and homework problems regarding these were hinted at. The questions are:

- If some players cooperate, can they make p_1 look like a liar? How many are required to do so?
- If some players cooperate, can they change the secret? How many are required to do so?

As an example for the second question, assume that we have 5 players, $t = 4$ and that players p_4 and p_5 are bad and want to change s to $s + 1$. p_4

and p_5 select the polynomial $P'(x)$ of degree 3 where $P'(\alpha_i) = 0$ for $i = 1, 2, 3$ and $P'(0) = 1$. They then report their secrets as $P(\alpha_i) + P'(\alpha_i)$, and the secret has thus been changed to $s + 1$.

7 Multi-party computations

Using Shamir's secret sharing, we can perform multi-party computations. In this scenario, we have some function $f(X_1, \dots, X_n)$ where each participant P_i has a part of the input X_i . We wish to calculate $f(X_1, \dots, X_n)$ and give the result to all participants without any participant learning anything about the other participants' inputs (except what may be learned from the function value, e.g. if $f(X_1, \dots, X_n) = X_1$, all participants would of course learn X_1).

We assume that participants are honest, but curious. That is, all participants will follow the protocol, but they will draw all logical conclusions from what they see. We also assume that we have private channels between each pair of players. These can be realized either as physical channels or by using encryption.

7.1 Linear functions

First, we discuss how to do this for a special case, a linear function $f = \sum_{i=1}^n f_i X_i$ over some field F . Each participant P_j is given a random point $\alpha_j \in F$ where $\alpha_i \neq \alpha_j, i \neq j$ and $\alpha_j \neq 0$. These points α_j are made public.

Each participant P_i uses Shamir's secret sharing to give out his secret X_i to the other participants, so P_i creates the polynomial $q_i(x) = \sum_{j=0}^{t-1} a_{j,i} x^j$ with $a_{0,i} = X_i$. Using the private channel, he sends $q_i(\alpha_j)$ to every other participant P_j .

Now consider the new polynomial $q(x) = \sum_{i=1}^n q_i(x) f_i$. Every participant P_j can calculate $q(\alpha_j)$ locally and then broadcast the result. The constant term $q(0) = \sum_{i=1}^n f_i X_i = f(X_1, \dots, X_n)$, which can be recovered by interpolation after t broadcasts of $q(\alpha_j)$'s.

Note that this does not seem to work for $n = 2$ as it seems to reveal too much information. For a linear function f , however, knowing $f(X_1, X_2)$ and either of X_1 or X_2 , the other input can always be calculated. Thus in fact the output reveals both inputs and thus a correct protocol for this situation in our model is that one player reveals his input and the other player gives the result.

Note that the field F must be large enough that there are enough random points in the field to give each participant a unique α_i . If the field is too

small, i.e. has fewer than $\leq n$ elements, it must be extended to a larger field. In particular, for a boolean function, $\text{GF}[2^{\lceil \log(n+1) \rceil}]$ would be a suitable field.

7.2 Arbitrary boolean circuits

To be able to do arbitrary boolean circuits, it is enough to show that it is possible to do some set of boolean functions which can be used to build up all other boolean functions. One such set is the one consisting of the gates **and** and **xor**. The scheme presented here is not very efficient, it uses n multi-party computations of linear functions for *every* **and**-gate in the circuit.

We will do calculations gate by gate and use Shamir's secret sharing to share the output of each gate over a field of the form $\text{GF}[2^\ell]$ for $\ell \geq \log(n+1)$. We assume that the inputs to the gate are shared and show how to share the output of the gate without leaking any information. At the end, the secrets can be pooled and the output of the circuit interpolated as usual.

Let $q_i(x) = \sum_{j=0}^{t-1} q_{i,j} x^j$ where $q_{i,0}$ is the value of the circuit at point i in the circuit. The **xor** gate is easy and can be done locally. Assume we have Shamir-shared inputs to the **xor**-gate, $q_1(\alpha_j)$ and $q_2(\alpha_j)$. Since **xor** is a linear function, every participant j can simply calculate $q_3(\alpha_j) = q_1(\alpha_j) + q_2(\alpha_j)$.

The **and**-gate is a bit harder. Given that we have Shamir-shared inputs to the **and**-gate $q_{1,0}$ and $q_{2,0}$ we want to Shamir-share $q_{3,0} = q_{1,0} \wedge q_{2,0}$. One idea may be to have each player simply calculate $q_3(\alpha_j) = q_1(\alpha_j)q_2(\alpha_j)$ which gives the correct $q_{3,0}$. However, doing so will make q_3 have degree $2t-2$, and after a few repetitions, interpolation will no longer be unique and the secret would be lost. Here, we assume that $2t-2 < n$. Another problem is that the coefficients of the new, longer q_3 are no longer randomly distributed.

Going from $q_3(x) = \sum_{i=0}^{2t-2} a_i x^i$ to $q'_3(x) = \sum_{i=0}^{t-1} a_i x^i$ (i.e. throwing away the high-degree terms of the multiplied polynomial) would alleviate the problem with the degree of q_3 . This transformation is a linear function, which we already know how to do. So, each participant calculates $q_3(\alpha_j) = q_1(\alpha_j)q_2(\alpha_j)$ locally. When this is done, all players cooperate (using the method to do a general linear function) to calculate $q'_3(\alpha_j)$ for each participant j , with the slight difference that instead of broadcasting the result, all players send the result to player j privately.

The problem that the coefficients of q_3 (and thus q'_3) are no longer randomly distributed still remains, however. One solution to this problem is to have each participant select a random polynomial $r_j(x)$ with degree $t-1$ and $r_j(0) = 0$ (otherwise, the shared secret would change). The participant j then proceeds to send $r_j(\alpha_i)$ privately to participant i , for every participant i . These are added to q'_3 and we use $q''_3(x) = q'_3(x) + \sum_{i=j}^n r_j(x)$.