CS 810: Complexity Theory		2/23/2007
	Lecture 16: Error-Correcting Codes	
Instructor: Dieter van Melkebeek		Scribe: Matt Elder

Last lecture, we constructed a family of "quick" pseudo-random generators, based on the assumption that there exists a language $L \in \mathrm{E}$ with high average-case hardness. In this lecture, we extend these results; we show that the worst-case hardness at length m, $C_L(m)$, can be substituted for the average-case hardness at length m, $H_L(m)$.

For most of this lecture, and almost all of the next, we prove this fact via error-correcting codes. Such codes will let us construct a language L' with the average-case hardness very close to the worst-case hardness of a given language L.

1 Circuit Lower Bounds Yield Pseudo-Random Generators

Last time, we found that we could generate efficient pseudo-random generators. More precisely, suppose we have a function $f : \{0,1\}^m \to \{0,1\}$ and an $(m, \log r)$ design $S_1, \ldots, S_r$ over $[\ell]$. We construct the function $G_r : \{0,1\}^\ell \to \{0,1\}$, where $G_r(\sigma) = (f(\sigma|_{S_1}), \ldots, f(\sigma|_{S_r}))$. The key property of G_r is that if $H_{L(f)}(m) \geq r^2$, then G_r is a $(1/r)$-PRG for circuits of size at most r. We also demonstrated a very efficient design when $\ell = O(m^2)$, such that G_r is "quick" - that is, in time linear-exponential in ℓ if $L(f) \in \mathrm{E}$.

If a hard language exists that allows us to construct such a pseudo-random generator, and we have a BPP algorithm A for a problem, then we can use the following deterministic algorithm A' to solve the same problem.

1. Construct the PRG G_r, which takes a random seed of length ℓ and outputs a pseudo-random string of length r.
2. For every seed $\sigma \in \{0,1\}^\ell$, run A, replacing its random bits with $G_r(\sigma)$.
3. Accept if A accepts on most seeds; reject otherwise.

If the complexity of A is small enough that the PRG G_r is known to fool it, then A' is a deterministic algorithm for the problem that A solves. Thus, we can get the following results:

1. If there exists a language L in E such that $H_L(m) \geq m^{\omega(1)}$, then we can build G_r that takes $r^{o(1)}$ input bits, so BPP $\subseteq$ SUBEXP.
2. If there exists a language L in E such that $H_L(m) \geq 2^{m^{\Omega(1)}}$, then we can build G_r that takes $(\log r)^{O(1)}$ input bits, so BPP $\subseteq \mathrm{DTIME}(n^{(\log n)^{O(1)}})$.
3. If there exists a language L in E such that $H_L(m) \geq 2^{\Omega(m)}$, then we can build G_r that takes $O(\log r)$ input bits, so BPP $\subseteq$ P.

In implication 3, we need a $(c \log r, \log r)$ design with $\ell = O(c^2 \log r)$ to show that we can build a G_r that takes $O(\log r)$ input bits. We assert that such a design exists, but omit its construction here. As an aside, implication 3 implies the following corollary.

Corollary 1. *BPP* $\subseteq$ *ZPP*$^{\mathrm{NP}} \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

Proof. By definition, a language is in BPP if it has a poly-time, bounded-error, randomized algorithm. As we've just shown, such algorithms can be derandomized using any function $f\colon \{0,1\}^m \to \{0,1\}$ in E that has hardness $H(f) \geq 2^{\Omega(m)}$.

So, consider χ_f, the characteristic string of the function f. The length of χ_f is 2^m. By a counting argument on the possible characteristic strings of f, we know that most functions on m bits have hardness $H(f) \geq 2^{\Omega(m)}$. So, given a BPP algorithm A, consider the following algorithm in ZPP$^{\mathrm{NP}}$:

1. Pick a random function $f\colon \{0,1\}^{O(\log r)} \to \{0,1\}$
2. Ask the NP oracle if there exists a circuit of size less than r that computes f. Because f is a function over $O(\log r)$ bits, this is an NP predicate. If there is a small circuit for f, halt and return "?".
3. Otherwise, construct a pseudo-random generator G built from the hard function f. Run the derandomized version of A using G. Accept if it accepts and reject if it rejects.

Since most candidates for f are hard languages, this algorithm halts and fails with low probability; in all other cases, it outputs the correct answer, so this is a ZPP$^{\mathrm{NP}}$ algorithm. So, if a language can be solved by an algorithm in BPP, then it can be solved in ZPP$^{\mathrm{NP}}$. Thus, BPP $\subseteq$ ZPP$^{\mathrm{NP}}$. ☐

Next we discuss the converse of these implications; instead of showing that hard languages imply PRGs, we show that PRGs imply hard languages.

2 Pseudo-Random Generators Yield Circuit Lower Bounds

Theorem 1. *If there exists a ϵ-PRG G computable in* E *that fools circuits of size at most r, then there exists a language $A \in$* E *with circuit complexity greater than r.*

Proof. The idea of this proof is that, given a PRG G, we can construct a language that differentiates between the distribution that G generates and the uniform distribution. Since G is a PRG, this language must have a high circuit complexity.

First, note that the length of the output of G must be larger than the length of its input, and $\epsilon < 1$; the existence of G otherwise is trivial. By only looking at the first $\ell + 1$ output bits of G, we can assume that G takes ℓ bits to $\ell + 1$ bits. Let A be the language $\left\{G(\alpha) \mid \alpha \in \{0,1\}^\ell\right\}$.

Suppose circuit C decides language A. Then,

$$\mathbf{Pr}_{\sigma \in \{0,1\}^\ell}\left[C\left(G(\sigma)\right) = 1\right] = 1, \text{ and}$$
$$\mathbf{Pr}_{\sigma \in \{0,1\}^{\ell+1}}\left[C(\sigma) = 1\right] \leq \frac{1}{2}.$$

Above, the first line holds by the definition of C. The second line holds because G maps its 2^ℓ inputs to no more than 2^ℓ distinct outputs. Together, these statements show that C differentiates between the distribution generated by G and the uniform distribution. Thus, since G is known to be undifferentiable by any circuit of size no more than r, the circuit C has size greater than r. This is true for any circuit that decides A, so the language A has circuit complexity greater than r.

To determine if $x \in A$, the naive algorithm is to compute $G(\sigma)$ for all seeds σ and see if x is the output of one of these. As we assume $G \in \mathrm{E}$ and must cycle over 2^ℓ seeds, this computation takes time $2^{O(\ell)}$ - showing that $A \in \mathrm{E}$. $\square$

So, we now have implication both ways: there exist quick PRGs that fool circuits of size r iff there exist functions in E that have circuit complexity greater than r.

3 Error-Correcting Codes

Informally, an *error-correcting code* is a function from a set of *information words* to a set of *codewords* such that, given a codeword with some not-too-large fraction of its bits flipped, the information word can be retrieved. Thus, error-correcting codes encode information in a manner robust against a certain amount of error.

We will use error-correcting codes to relate a language L' that has high average-case circuit complexity to a language L with high worst-case circuit complexity. Suppose that $X_{L|_m}$ is the characteristic sequence of L on inputs of length m, that is, the i^{th} bit of $X_{L|_m}$ is 1 iff the i^{th} m-bit word is in the language L. Now, for some m', let L' be the language whose characteristic sequence on inputs of m' bits is the codeword corresponding to the information word $X_{L|_m}$ in some error-correcting code.

We would like to be able to conclude that if L is worst-case hard for circuits of size s, then L' is average-case hard for circuits of slightly smaller size. The natural way to try to prove this is the following. Suppose we have a circuit C of size s' that computes L' on a large fraction of inputs. Given an input x that we want to compute L on, we want to use C in addition to the decoding procedure for the error-correcting code to compute $L(x)$. The goal is to accomplish this with only a small increase in the size of the circuit. An examination of this process (to be carried out in the next lecture) reveals that the error-correcting code needs to satisfy the following properties:

1. The ECC can handle error rates as high as $1/2 - \epsilon$ for small ϵ.

2. The ECC employs *local decoding*; that is, a single bit of the information word can be derived by examining only a small portion of the codeword.

3. The ECC can encode in polynomial time. (This way, $L \in \mathrm{E}$ implies $L' \in \mathrm{E}$, and $m' = cm$ for some positive constant c.)

Of course, to usefully discuss these properties, we must formally define error-correcting codes. We give appropriate definitions and then consider a number of error-correcting codes to see if we can find one that meets the above goals.

Definition 1. *An* (N, K, D) error-correcting code *(ECC) over the alphabet* Σ *is a function* $E : \Sigma^K \to \Sigma^N$ *such that for all distinct* $x_1, x_2 \in \Sigma^K$*, the Hamming distance*[1] *between* $E(x_1)$ *and* $E(x_2)$ *is greater than* D*.*

Suppose we process a word $x \in \Sigma^K$ with an ECC, producing $y = E(x)$. Suppose that uncontrolled conditions flip some bits of the code word at random, producing the *received* word $z = \mathrm{perturb}(y)$ and destroying our memory of y. If fewer than D bit flips have occurred between

[1]The Hamming distance between two binary strings (of equal length) is the number of bits in which they differ.

y and z, then z cannot be any legal codeword, so we can detect that bits have been changed. If fewer than $\lfloor (D-1)/2 \rfloor$ bit flips have occurred, then in terms of Hamming distance, z is closer to y than it is to any other legal codeword, so it can be corrected.

The parameters of an ECC that we wish to optimize are:

- The *relative distance* $d = D/N$, which we want to make large.
- The *rate* K/N, which we want to make small.
- The complexity of encoding and decoding, which we want to keep low.

The rate and relative distance are, in a sense, opposed to each other. For example, a very small rate restricts the minimum Hamming distance between two codewords, as it leaves little extra space in Σ^N.

Many of the codes we are interested in belong to a general class of ECCs, called linear error-correcting codes. Intuitively, each codeword bit is a linear function of the message bits. This is define formally as follows.

Definition 2. *An ECC is* linear *if its range is a linear subspace with rank K of $GF(q)^N$ for some prime power q. We designate a linear ECC with square brackets, thus: $[N, K, D]_q$.*

A linear ECC has some nice properties, such as having a generator matrix, which we will not further discuss here. Next, we give several examples of linear ECCs.

3.1 Hadamard Code

The Hadamard code, with $q = 2$, takes the information word $a \in \{0,1\}^K$ to the codeword $(a \cdot x)_{x \in \{0,1\}^K}$. That is, the concatenation of the dot products of a and x across all possible binary words x of length K. For this code, $N = 2^K$ and $d = 1/2$. This relative distance is very good, but the rate is very bad.

Consider how we might decode a received word, after it has been encoded and mangled. Let $r(x)$ be the piece of the received word in position x. Suppose that $\Pr[r(x) = a \cdot x] \geq 3/4 + \epsilon$. If we want the bit a_i, then we can look at two values of x that differ at bit i. We pick x uniformly at random, and look at this bit as well as the bit $x \oplus e_i$. Then:

$$\begin{aligned}
&\Pr[r(x) \oplus r(x \oplus e_i) = a_i e_i] \\
\geq &1 - (\Pr[r(x) \text{ is corrupted}] + \Pr[r(x \oplus e_i) \text{ is corrupted}]) \\
\geq &1 - 2(\frac{1}{4} - \epsilon) \geq \frac{1}{2} + 2\epsilon.
\end{aligned}$$

Thus, the probability that we can decode a single bit by examining just two others is greater than 1/2. So, as long as the error rate of transmission is slightly below 1/4, we can correct all errors. Since the relative distance of any ECC is no more than 1/2, the error-correction rate that this algorithm yields is arbitrarily close to the best rate we can ever have.

Notice that the Hadamard code fails to meet our original goals: although local decoding is possible, the encoding (and therefore time to encode) is exponentially long, and we can handle error rates only up to 1/4.

3.2 Reed-Solomon Code

The Reed-Solomon code transforms the information word $a \in GF(q)^K$ to the codeword $(P(x))_{x \in GF(q)}$, where P is the polynomial of degree $K-1$ whose coefficients are the "digits" of a (i.e. $a = (a_0, a_1, ..., a_{K-1})$ and $P(x) = a_0 + a_1 x + ...a_{K-1}x^{K-1}$). This code relies on the fact that different polynomials of degree $K-1$ can have at most $K-1$ points of intersection, meaning the distance between distinct codewords is at least $N-K$.

The parameters of the Reed-Solomon code are $N = q$, $K \leq q$, and $d \geq 1 - K/N$. Because the Reed-Solomon code yields an "adjustable" trade-off between the rate and the relative distance, and is fairly easy to encode and decode, it's heavily used in many communications and data-retrieval applications.

But can we use the Reed-Solomon code for the current purposes - does it meet the three goals stated earlier? By setting $K = N2\epsilon$, the code can handle errors up to $\frac{1}{2} - \epsilon$, and both encoding and decoding can be performed in polynomial time. However, decoding is very nonlocal - essentially the entire received word must be examined to retrieve a single bit of the message.

3.3 Reed-Müller Code

The Reed-Solomon code fails for our purposes because the decoding procedure is inherently non-local. The Reed-Müller code corrects this problem by using multi-variate rather than univariate polynomials. We will see in a moment how to take advantage of multi-variate polynomials to perform local decoding. First, consider the representation of the message. In the Reed-Solomon code, we represent the message as the coefficients for a polynomial. For the Reed-Müller code, we will think of the message as an m-variate polynomial with individual degrees less than s. We could use a similar encoding as was used for the Reed-Solomon code: interpret the message as the coefficients of the polynomial. However, this encoding inherently leads to non-local decoding - to recover the coefficients we need to recover the entire polynomial, meaning we would need to examine a large fraction of the received word bits.

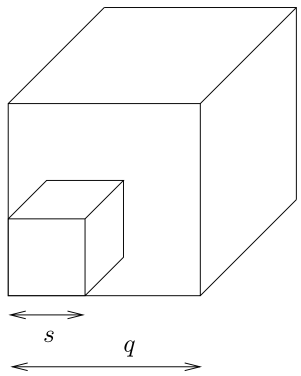

Figure 1: An illustration of the scheme used for the Reed-Müller code for the case of $m = 3$. The message gives the evaluation of a polynomial on a sub-cube of size s^m, and the encoding is the evaluation of an interpolating polynomial on the entire space.

We use a different encoding that will allow local decoding. The information word a is composed of s^m elements from $GF(q)$. We view each element as the evaluation of some m-variate polynomial P on a particular element of $GF(q)^m$ taken from a sub-cube of size s^m. This is illustrated in Figure

1. Encoding is performed by determining the polynomial P from the points given in the message, and then evaluating P on all possible m-tuples over $GF(q)^m$.

It is immediate that for the Reed-Müller code, $K = s^m$ and $N = q^m$. By the Schwartz-Zippel Lemma, the code achieves relative distance $d = 1 - \frac{ms}{q}$.

3.3.1 Encoding

We have specified the correct encoding of the Reed-Müller code above, but it is not immediate that this can be done efficiently: we need to determine the unique polynomial P with each individual degree less than s that interpolates a given set of values on a sub-cube of size s^m. Let $a_i \in GF(q)$ denote the i^{th} value of a. Let ϕ be some invertible map between the integers 1 through s^m and a subcube of $GF(q)^m$ of size s^m. We define want to determine P so that $P(\phi(i)) = a_i$.

We show that we can construct P by building polynomials that interpolate each element of the sub-cube. The polynomial δ_c is defined for an m-variable constant vector c, which is inside the s^m-subcube. We want $\delta_c(c) = 1$ and $\delta_c(x) = 0$ for all other values x inside the s^m-subcube. We can construct δ_c as follows:

$$\delta_c(x) = \alpha \prod_{i=1}^{m} \prod_{\substack{j \neq c_i \\ 0 \leq j < s}} (x_i - j).$$

Here, α is just a normalization constant; we use this so that $\delta_c(c) = 1$. For all values of $x \neq c$ in the s^m-subcube, there is some index i such that $x_i \neq c_i$. By the construction, there must then be a multiplicand in $\delta_c(x)$ of the form $x_i - x_i$, so $\delta_c(x) = 0$. Note that the degree of δ_c in each variable is less than s.

So, we can compose the desired polynomial P as a linear combination of the polynomials δ_c:

$$P(x) = \sum_{i=1}^{s^m} a_i \delta_{\phi(i)}(x).$$

As the degree of each δ_c in each variable is less than s, the same is true of P. We conclude that this formula determines the unique such polynomial interpolating the given points.

Once we have constructed P from the information word a, we produce the codeword b by concatenating the value of $P(x)$ for all values of x in $GF(q)^m$. As an aside, we point out that the the codeword contains an exact copy of the message a (the portion of the codeword where we evaluate P on the elements of the sub-cube that were used to construct P). Codes with this property are called *systematic codes*.

3.3.2 Decoding

To decode, we could query the received word at every position and construct a polynomial that differs from the received word in the fewest number of positions. We do not do this as our goal is to perform as few queries of the received word as possible. Recall that each element of the message corresponds to the evaluation of P on some point in the s^m-subcube. The basic idea is to look at P restricted to a random line through the point corresponding the position in the message we want to decode. P restricted to this line is a univariate polynomial, so if the received word agrees with P on a large fraction of the points on the line, we can recover the values of P on the line (including the point we are interested in). The construction is illustrated in Figure 2.

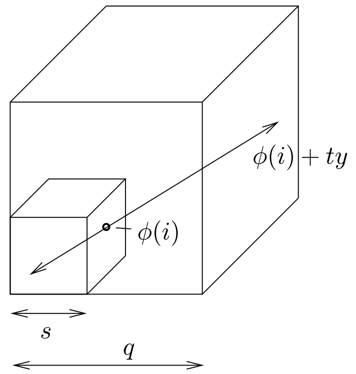

Figure 2: Reed-Müller decoding. To determine $a_i = P(\phi(i))$, construct a line $\phi(i) + ty$ through $\phi(i)$ for a randomly chosen y, then determine the univariate polynomial $P(\phi(i) + ty)$. For $t = 0$, this gives $P(\phi(i))$.

We now formalize the decoding procedure. Suppose r is our received word. Let $r(x)$ denote the value of r in the position corresponding to $x \in GF(q)^m$; and let $P(x)$ denote the analogous position in the correct codeword. To retrieve the i^{th} digit of the message a, we would like to determine the value of $P(\phi(i)) = a_i$. To determine $P(\phi(i))$, we pick a random point $y \in GF(q)^m$, select ms distinct values for $t \neq 0$, and examine $r(\phi(i) + ty)$ at each one. We know that $P(\phi(i) + ty)$ is a polynomial in t of degree at most ms. Suppose $P = r$ for the values we have chosen. In this case, we can determine the polynomial $P(\phi(i) + ty)$, and evaluate it at $t = 0$ to get $P(\phi(i)) = a_i$.

So if $P = r$ on the chosen points, we recover the correct value. We now bound the probability that $P \neq r$ for at least one of the chosen points. Suppose $r(x)$ differs from $P(x)$ in at most γ fraction of locations. As each point we query r on is uniformly distributed, the probability that $r(x) \neq P(x)$ for each query x is at most γ, by our above assumption. So, the probability that $r(\phi(i)+ty) \neq P(\phi(i)+ty)$ for some selected value of t among the $m(s-1)$ values that we selected is at most $(ms)\gamma$. If we assume that $\gamma \leq \frac{1}{3ms}$, the probability we output an incorrect value for a_i is at most $1/3$. Furthermore, we can repeat the above process for different random values of y, and then take the majority vote for the value of a_i. In this way, we can make this decoding algorithm work with probability $1 - \epsilon$ for ϵ as small as we like.

The above analysis only works provided the error rate in transmission is no greater than $1/(3ms)$. Essentially the same techniques can be used with higher error rates, but we stuck to lower error rates in the above to keep the analysis simple. A decoding algorithm for the Reed-Müller code which corrects higher errors is given on Page 14 of [1].

We also point out that the decoding procedure is randomized, whereas we often want to have a deterministic decoding procedure. Since we will have circuits perform the decoding procedure in our worst-case to average-case reduction, requiring randomness will not be a problem - we will be able to hardwire "good" random bits into the circuits.

3.3.3 Parameters

We would like to set the parameters of this code in such a way to meet the three conditions we originally set out to meet. We already have that $N = q^m$, $K = s^m$, and $d = 1 - \frac{sm}{q}$. We want to choose s, m, and q, so that: i) d is close to 1, meaning we can correct even with error rates close to

$\frac{1}{2}$, ii) ms is small, so the number of queries in the decoding procedure is small, and iii) $N = K^{O(1)}$, so the encoding is polynomially long (and thus also polynomial-time computable).

We point out that for d to be positive, we need $sm < q$. Then iii) implies that

$$K^{O(1)} \geq N = q^m \geq (ms)^m = K \cdot m^m,$$

so $m^m \leq K^{O(1)}$. Taking logarithms, we have that $m \log m \leq O(\log K)$, and therefore $m \leq O(\frac{\log K}{\log \log K})$.

ii) combined with the constraint that $K = s^m$ means that making m as large as possible will minimize sm, so we set $m = \Theta(\frac{\log K}{\log \log K})$. As $K = s^m$, this means that $s = \Theta(\log K)$.

We have set the parameters so that we get a code with polynomial stretch and requiring only a poly-logarithmic number of queries to locally decode. This is almost good enough for what we want to do. There are two issues that still need to be dealt with: the code is not binary, and (we will see) the distance is not good enough.

3.4 Concatenation of Reed-Müller and Hadamard

To deal with the problem of the Reed-Müller code being a non-binary code, we concatenate it with the Hadamard code. If we start with an $[N, K, d]$ code over an alphabet of q elements, concatenation with the Hadamard code yields a binary $[Nq, K \log q, \frac{d}{2}]$ code. This concatenated code has all of the properties that we want except that the distance is not good enough. Recall that we wanted to be able to handle error rates close to $\frac{1}{2}$. To do this with unique decoding, we need the distance $\frac{d}{2}$ to be close to 1. This is not possible since $d \leq 1$.

In the next lecture, we will see how to get around this problem by relaxing the requirements of uniquely decoding the correct message. The relaxed notion is called *list-decoding*, where we require the decoding procedure to output all messages whose encoding is close to the received word. In the next lecture we will see a list-decoding procedure for the concatenated Reed-Müller/Hadamard code that satisfies all three of our original goals.

References

[1] Madhu Sudan, Luca Trevisian, and Salil Vadhan. *Pseudorandom Generators without the XOR Lemma*. J. of Computer and System Sciences, 62(2):236-266, 2001. Preliminary version: Proc. of 31st ACM STOC, 1999. Accessed at `http://www.cs.berkeley.edu/%7Eluca/pubs/`