CS 710: Complexity Theory 11/15/2011

Lecture 20: Error Correcting Codes

Instructor: Dieter van Melkebeek Scribe: Xi Wu

From the last two lectures we have seen that a language that is hard "on average" implies efficient derandomization of BPP. In this lecture we begin the development of replacing the requirement of "average-case hardness" by the "worst-case hardness". Since requiring worst-case hardness is weaker than requiring average-case hardness, this gives us more plausible derandomization results of BPP. The way we achieve this is through error correcting codes (ECC). We give some intuition of how to use ECCs for our purposes, the properties we need the ECC to satisfy, and discuss pros and cons of various classical ECCs w.r.t. fully satisfying these properties.

1 Weaker Assumption for Derandomizing BPP?

In the previous lecture we discussed conditional results on time bounded derandomization. We showed how to transfer circuit lower bounds into derandomization of BPP. To remind you of these results, we define $H_L(n)$ to be the largest s such that circuit of size s cannot correctly compute more than $\frac{1}{2} + \frac{1}{s}$ fraction of inputs of length n. This notion characterizes the average hardness of L. The main theorem we proved last time is as follows:

Theorem 1. *If there exists $L \in \mathrm{E}$ and that $H_L(m) \geq r^2$ then there is a quick $\frac{1}{r}$-PRG of seed length $\ell(r) = O(m^2)$ for circuit of size at most r.*

By "quick" it means that the generator runs in time *linear exponential* in seed length $\ell = \ell(r)$ (i.e. $2^{O(\ell(r))}$). Recall how the generator G_r works. It generates r pseudorandom bits by concatenating $L(\sigma|_{S_i})$ for $i \in [r]$ where σ is the seed of length ℓ and $S_1, S_2, \ldots, S_r \subseteq [\ell]$ are subsets of size m where the size of pairwise intersection is bounded by $\log r$. Finally it turns out that $S_1, \ldots, S_r$ can be constructed efficiently with $\ell = O(m^2)$ for all r and $m \geq \log r$.

Now initiating L with different average hardness gives different results of derandomizing BPP.

1. If $(\forall c > 0)(\exists L \in \mathrm{E}) H_L(m) \geq m^c$ then $\mathrm{BPP} \subseteq \cap_{\epsilon>0} \mathrm{DTIME}(2^{n^\epsilon}) = \mathrm{SUBEXP}$.
2. If $(\exists L \in \mathrm{E}) H_L(m) \geq m^{\omega(1)}$ then $\mathrm{BPP} \subseteq \mathrm{DTIME}(2^{n^{O(1)}})$.
3. If $(\exists L \in \mathrm{E}) H_L(m) \geq 2^{m^{\Omega(1)}}$ then $\mathrm{BPP} \subseteq \mathrm{QP} = \cup_{k\in\mathbb{N}} \mathrm{DTIME}(2^{\log^k n})$.
4. If $(\exists L \in \mathrm{E}) H_L(m) \geq 2^{\Omega(m)}$ then $\mathrm{BPP} = \mathrm{P}$.

Let's take the last statement as an example of how the derandomization works. Let L be the hard language where $H_L(m) \geq 2^{\Omega(m)}$, say $H_L(m) = 2^{\frac{1}{2}m}$. Fix an arbitrary BPP language L' which requires circuit size r. To derandomize it note that $m = c \log r$ for some constant c. By picking some sufficiently large constant c, in this case $c \geq 4$, it ensures that $H_L(m) \geq 2^{\frac{c}{2}\log r} = r^{\frac{c}{2}} \geq r^2$. By Theorem 1 we have G_r is an $\frac{1}{r}$-PRG with seed length $O(m^2) = O(\log^2 r)$. This puts BPP is $2^{O(\log^2 r)} = r^{O(\log r)}$ by enumerating all seeds, but this is not polynomial time in r. Fortunately it turns out that for $m = c \log r$, there exists $(m, \log r)$-design with $\ell(r) = O(c^2 \log r)$. This allows enumerating all seeds in time polynomial in r, thus putting L' in P.

One unsatisfying point of our current state is that requiring average-case hardness is a somewhat strong condition. What we are going to develop in the following two lectures is to replace average-case hardness by worst-case hardness, in the following sense. Given a language $L' \in$ E of worst-case hardness, we construct, via ECC, another language $L \in$ E that is hard on average. This would relax our requirement of average-case hardness to worst-case hardness where it makes our conditional results of derandomizing BPP more plausible. Before further development, we remark that the *construction* of theorem 1 gives us a new proof that $\text{BPP} \subseteq \text{ZPP}^{\text{NP}} \subseteq \Sigma_2^p \cap \Pi_2^p$.

Corollary 1. $\text{BPP} \subseteq \text{ZPP}^{\text{NP}}$.

Proof. Consider the BPP algorithm of circuit size r. If we have a language $H_L(m) = 2^{\Omega(m)}$ then we can directly use the above generator to derandomize it in time polynomial in r. The difficulty is that how can we find such L.

The key observation is that we only need $m = O(\log r)$, therefore we can guess the characteristic sequence of a Boolean function f at length m, which is of $2^{O(\log r)} = \text{poly}(r)$ bits. Using probabilistic method, it is straightforward to verify that a random Boolean function at length m has the property that $H_f(m) = 2^{\Omega(m)}$ (actually most of them do).

Now the base ZPP machine does the following, it guesses the characteristic function of a Boolean function f at length m, and makes a coNP oracle query to to verify that no circuit of size r can approximate f on more than $\frac{1}{2} + \frac{1}{2^{\Omega(m)}}$ fraction of inputs. W.h.p this is the case and then it continues to do derandomization, otherwise it outputs an "?" saying "unknown". $\square$

2 Outline from Average-Case Hardness to Worst-Case Hardness

Our goal is to construct out of a worst-case hard language L' an average-case hard language L. Note that for worst-case hardness $C_{L'}(m) = s$ means $(\forall C, |C| < s)(\exists x, |x| = m)[C(x) \neq L(x)]$, and it might be very well possible that a circuit of size $< s$ would fail on very few points. By taking contraposition, what we would like to have is a statement like the following:

We construct L such that, if we can compute L correctly only on some fraction of inputs, or equivalently approximate L well enough, then we can compute L' on every input with a slightly larger circuit. However now, because computing L' everywhere has a circuit lower bound, then there is a circuit lower bound on computing L correctly on certain fraction of inputs.

Now it is somewhat natural to view the membership string χ' of L' at length m' as an information word, and we encode it, using ECC, as the membership string χ of L at some length m. Because of the nature of ECC, we would have the claim because if we can approximate χ on sufficiently large fraction of points, then χ' can be recovered and thus we can correctly compute L' everywhere: now the worst-case circuit lower bound of L' would give us an average-case circuit lower bound of L for doing this! However, simply using naive ECC to encode χ' is not enough. Because what we want is actually in some "local" sense: compute $L'(x')$ for some x' of length m', rather than computing the characteristic sequence of L' at length m'. Using naive encoding, we will end up looking at an encoding of a bit string of length $2^{m'}$, which is exponentially larger than $|x'|$. In summary we would like the ECC to satisfy the following properties:

1. It needs to be a binary code, as χ and χ' are characteristic sequences of L and L',

2. ECC can handle errors up to $\frac{1}{2} - \frac{1}{H_L(m)}$. Because we want lower bounds even when we can approximate $\frac{1}{2} + \frac{1}{H_L(m)}$ fraction of the inputs.

3. Local decodability. Suppose c is an encoding of s, and we only want to know $s[i]$. This shall be done by only querying few locations of c, rather than looking at all of them. This implies that the decoding must be randomized, otherwise we cannot handle that much error.

4. ECC is computable in time $\text{poly}(2^{m'})$. This is because we need $L \in \text{E}$ so that the derandomization with average-case hardness can be applied. More precisely, if $L' \in \text{E}$ and the encoding can be done in time polynomial in $2^{m'}$, then we can compute $L(x)$ by encoding L' at m' and then extract the bit at position x, which takes time linear exponential in m. Note that this condition indicates that $\text{poly}(2^{m'}) \geq 2^m$ and so $m = O(m')$.

With these requirements in mind, now we consider classical error-correcting codes to see whether one of them could satisfy precisely all these. Unfortunately none of them does: each of them is good at some properties, but fails on some other. It turns out that the final solution that works is to *combine* two of them and use *list decoding*. This will be the topic of the next lecture.

3 Error Correcting Codes

An (N, K, D)-code over alphabet Σ is a mapping $E : \Sigma^K \mapsto \Sigma^N$ such that $(\forall x, y \in \Sigma^K)$ with $x \neq y$, $d_H(E(x), E(y)) \geq D$, where d_H denote the relative hamming distance of two code words. Note that any encoding function is injective because in our setting we want different messages to have different encodings. Here we use capital letters because N, K are indeed exponentially larger than the parameter we are interested in: the complexity measure is the length of a single input m, while the codeword is of length 2^m as there are 2^m input strings at this length. A code has the following important parameters:

1. Code rate: $\frac{K}{N}$, this measures the redundancy we introduce into the code. We'd like this quantity to be as large as possible, so it will not blow up too much the size of the information word. The hope is that the rate approaches to 1.

2. Relative distance: $\delta = \frac{D}{N}$, this measures how much error we can detect or correct. Again we hope this to be large. Note that even with relative distance 1 we can only hope to correct half errors when using a binary alphabet.

3. The computational complexity of encoding and decoding.

Naturally, increasing code rate means introducing less redundancy which may in turn reduce the relative distance, thus we can deal with less errors. Conversely, increasing the relative distance will eventually require more redundancy thus decrease the code rate and increase the computational complexity of codes. These indicate that we have tradeoffs between the first and the second parameter, and these tradeoffs have impact on the third parameter.

A q-ary linear code of length N is a subspace of $\mathbb{F}_q^N$, where $\mathbb{F}_q = \text{GF}(q)$ is a field with q elements. In this case, a linear combination of codewords is still a codeword. We use $[N, K, D]_q$ to denote a linear (N, K, D)-code over $\Sigma = \mathrm{GF}(q)$. For $q = 2$ it is called a binary code, and for $q = 3$ a ternary code.

Let $\Sigma = \{0, 1\}$ be the binary alphabet. Suppose $y = E(x)$ and is sent over a noisy channel, and the receiver gets $z \neq y$ due to noise. Let's consider error detection and correction of the code. For error detection, we mean that the receiver can realize that something is wrong by looking at z. Because any two codewords are at Hamming distance at least D, so with at most $D - 1$ bits corrupted, we are sure z is not any valid codeword, and thus detect an error. If D or more bits are corrupted then one may end up with another codeword.

For error correction, we mean that the receiver can come up with the *unique* message x associated with y. Here the best thing we can do is to find the nearest valid codeword $\tilde{z}$ around z, and check out the message that $\tilde{z}$ encoded.

If the error is within $\lfloor \frac{D-1}{2} \rfloor$, then the Hamming ball centered at the received word of radius $\lfloor \frac{D-1}{2} \rfloor$ has at most one codeword, which gives unique decoding. Otherwise if z is already more than $\lfloor \frac{D-1}{2} \rfloor$ far away from y then by our assumption that D is the bound of Hamming distance between codewords, z may correspond to two or more codewords which encode different messages. This implies that unique decoding is possible up to Hamming distance at most $\lfloor \frac{D-1}{2} \rfloor$.

Now we give some concrete examples of error correcting codes.

3.1 Hadamard Code

Hadamard is a binary code. Given message $x \in \{0, 1\}^K$, we view it as the coefficients of a linear function $Y(a_1, \ldots, a_i) = \sum_{1 \leq i \leq K} x_i a_i, a_i \in \{0, 1\}$, and thus the codeword is the function Y, which is a 2^K-vector over GF(2). Precisely, the encoding function E sends x to

$$(\langle a, x \rangle)_{a \in (\mathrm{GF}(2))^K}$$

Clearly $N = 2^K$ and the rate is $\frac{K}{2^K}$ (terrible), and the relative distance is $\delta = \frac{D}{N} = \frac{1}{2}$, which is the best we can hope for. To see this suppose x and y differ at certain locations. Let $E(x)$ and $E(y)$ be corresponding encodings. We would like to show $\Pr_{a \sim \{0,1\}^K}[\langle a, x \rangle = \langle a, y \rangle] = \frac{1}{2}$. W.l.o.g suppose $x_1 \neq y_1$, then

$$\begin{aligned}
\sum_{1 \leq i \leq K} a_i x_i &= \sum_{1 \leq i \leq K} a_i y_i \\
\iff a_1 x_1 + a_1 y_1 &= \sum_{2 \leq i \leq K} a_i (x_i + y_i) \\
\iff a_1 (x_1 + y_1) &= \sum_{2 \leq i \leq K} a_i (x_i + y_i) \\
\iff a_1 &= \sum_{2 \leq i \leq K} a_i (x_i + y_i)
\end{aligned}$$

Note that x and y are fixed, so for any choices of $a = (a_1, \ldots, a_n)$, with probability $\frac{1}{2}$ one of two choices for a_1 makes the equality hold.

Let's consider the decoding of the Hadamard code. One nice property of the Hadamard code is that it's locally decodable. Given some encoding Y, and we want to know x_i. Because Y encodes a linear function with coefficients $x = (x_1, \ldots, x_K)$ we would consider $Y(e_i) = x_i$. However it might be very well possible $Y(e_i)$ is corrupted. We observe that Y is a linear function so $Y(e_i) = Y(a+e_i) - Y(a)$. Further because we are over GF(2), so $Y(e_i) = Y(a+e_i) + Y(a)$ for any a. So the following decoding algorithm intuitively works if a large fraction of encodings are still correct:

```
Algorithm 1: Local-Hadamard-Decoding
(1)    Pick a random $a \in \{0,1\}^K$.
(2)    Output $Y(a+e_i) + Y(a)$.
```

Suppose a fraction of δ bits is corrupted. So we will have an error if either $Y(a)$ or $Y(a+e_i)$ is incorrect. Because a and $a+e_i$ are both uniformly distributed among all points, so the error probability is bounded from above by 2δ. We need the error to bounded away below $1/2$ so $2\delta < 1/2$ thus $\delta < 1/4$. Setting $\delta = 1/4 - \varepsilon$, this gives error probability $\frac{1}{2} - 2\varepsilon$.

Although the Hadamard code is locally decodable, it fails our other requirements. First, it can only correct error up to $1/4$ and thus not a fraction $\frac{1}{2} - \frac{1}{H_L(m)}$. Second, the rate is too large: we want linear blowup while the Hadamard code gives exponential blowup. Of course, local decodability is desirable and actually we will use it as a building block in our final construction.

Next we introduce the so-called Reed-Solomon code which improves the rate using a larger field.

3.2 Reed-Solomon Code

Compared to the Hadamard code, the Reed-Solomon code uses a larger field GF(q) of some prime power q and views x as the coefficients of a univariate polynomial over GF(q) of degree at most $K-1$. Precisely we map x to $p(Z) = \sum_{1 \le i \le K} x_i Z^{i-1}$, where Z is a formal variable taking values in GF(q). We encode $P(Z)$ by the vector $(P(a))_{a \in \mathrm{GF}(q)}$. This indicates that $N = q$ and $K \le q$. Consider relative distance, any univariate polynomial of degree $K-1$ over GF(q) has at most $K-1$ roots and so $\delta \ge 1 - \frac{K-1}{q}$.

By controlling K, q we can full-fill the error-correction requirement of large fraction of errors. However there are two issues with Reed-Solomon code. First it is not a binary code. The other issue is that its *local-decodability* is not very good. Recall that local-decodability means that the amount of information used to recover one position should be much less than the amount of information used to recover the whole message. However even recovering one coefficient requires looking at K positions, which is already sufficient to recover the whole polynomial.

3.3 Reed-Müller Code

The final code we introduce is the Reed-Müller code, which uses multivariate polynomials. In a sense, Hadamard code fixes the number of variables, the function must be linear and the field size is 2. Reed-Solomon code fixes the number of variables to be 1, the degree and field size can vary, and Reed-Muller code allows all these three: number of variables, degree and field size to vary.

In a q-ary Reed-Müller code, the information word is interpreted as a low-degree polynomial in q variables and evaluating it at all q-tuples of field elements yields the encoding. Intuitively the information is packed on a cube of q dimensions. Why this would help local decoding? It turns out that for decoding one can draw a random line through the point in the cube where we want to evaluate the original polynomial, querying the received word at all points along this line to get a univariate polynomial, and then decode it using Reed-Solomon decoder. In this way, much less amount of information is needed compared to recovering the whole multivariate polynomial. More details will covered in the next lecture.

4 Next Time

Next time we will finish the discussion of Reed-Müller codes and present the worst-case to average-case reduction which combines Hadamard code, Reed-Müller code and list decoding.

Acknowledgements

In writing the notes for this lecture, I perused the notes for lecture 18 by Dmitri Svetlov from the Spring 2010 offering of CS 710, as well as lecture 16 by Matt Elder from Spring 2007 offering of CS 810.