CS395T: Pseudorandomness and Combinatorial Constructions
Lectures 20 and 21, March 29 and April 3, 2001
Lecturer: Dr. David Zuckerman. Scribe: Joel A. Tropp

1 Introduction

In this article, we prove the Hardcore Bit Theorem of Goldreich and Levin. This result can be used to establish the equivalence between hard functions and pseudorandom generators. The essential content of the theorem is that, given the image of a string under a one-way function, it is also hard to determine the dot-product of that string with a known random string.

First, we shall develop in detail the intuition behind the proof.

2 Worst-Case Hardness and Average-Case Hardness

The first question we ask is how to convert worst-case hardness to average-case hardness. Let us look at a simple example.

2.1 The Discrete Log Problem

Given a prime p, a field generator $g \in \mathbb{Z}_p^*$ and a field element $a \in \mathbb{Z}_p^*$, we wish to output an exponent k such that $g^k \equiv a \pmod{p}$. This is more or less equivalent to solving the equation $k = \log_g a$ over the field.

Now, fix a prime p and a generator g. Assume we can compute the discrete log for an ε-fraction of the field elements $a \in \mathbb{Z}_p^*$ using an efficient algorithm A. How can we develop a randomized algorithm such that the probability of correct computation is high for any field element a? The idea is to introduce a random catalyst so that we can convert any input a into an element for which we can correctly compute the discrete log.

Algorithm 2.1 (Discrete Log) *Given input $a \in \mathbb{Z}_p^*$, we output the discrete log of a with respect to a generator $g \in \mathbb{Z}_p^*$.*

1. Choose $\ell \in \mathbb{Z}_{p-1}$ uniformly at random.
2. Compute $a' = a\, g^{\ell}$.
3. Run algorithm A with input a' to obtain a candidate k' such that $g^{k'} \equiv a' \pmod{p}$.
4. Compute $g^{k'} \bmod p$ to ascertain if k' is the discrete log of a' with respect to g. We can easily make this determination in polynomial time. Since we have chosen ℓ uniformly at random and A outputs the correct answer for an ε-fraction of possible inputs, we see that k' is the discrete log of a' with probability ε.
5. If k' is incorrect, we return to step 2.
6. Otherwise, output $k = k' - \ell$.

We note that if $\varepsilon = 1/\mathrm{poly}(p)$, then the expected running time is polynomial in p. And we have shown that if discrete log is easy on average, then it is easy in the worst case. We believe that discrete log is hard in the worst case, so using the contrapositive we have supplied evidence that discrete log is hard on average.

The technique which we have illustrated here is called random self-reducibility. We seek to generalize this technique.

2.2 Errors in Polynomial Evaluation

Let A be a program which evaluates an unknown univariate polynomial of degree no more than d. This program, however, makes an error with a probability as large as $1/(3d)$. It is possible that some inputs will never yield the right answer. How can we mitigate this catastrophic behavior?

The solution is to reduce the problem to many random instances. Let's say we want to evaluate the polynomial at a point x. We choose d points $\{p_i\}$ not equal to x at random. Run A for each point p_i to obtain y_i. Find the unique polynomial P of degree d which passes through all the points (p_i, y_i), and evaluate P at the point x. Even if the random points are strongly correlated, we still have the error estimate

$$\Pr[\ A(x) \text{ is incorrect }] \leq d \Pr_x[\ A \text{ is in error }] = 1/3.$$

2.3 Worst-Case Hardness and Random Self-Reducibility

Now, we shall display an algorithm which can evaluate everywhere a function which is worst-case hard using an algorithm which can only evaluate that function in some parts of its domain.

Let $f : \{0,1\}^n \to \{0,1\}$ be a function which is hard in the worst case. We shall extend f to a randomly self-reducible function using its multilinear extension

Let $p \geq n+2$ be a prime, and write $\mathbb{F} = \mathbb{F}_p$. The multilinear extension $\hat{f} : \mathbb{F}^n \to \mathbb{F}$ satisfies

1. $\hat{f}|_{\{0,1\}^n} = f$ and

2. $\hat{f}$ is multilinear, i.e. affine in each variable.

By the first property, it is clear that $\hat{f}$ is as hard as f in the worst case. And to see that $\hat{f}$ is unique, observe that $\hat{f}(x_1, \ldots, x_n)$ can be determined uniquely from $\hat{f}(x_1, \ldots, x_{n-1}, 0)$ and $\hat{f}(x_1, \ldots, x_{n-1}, 1)$. So we can use recursion to compute the value of the function at any point.

Assume next that we have an efficient algorithm which can compute $\hat{f}$ for at least a $[1-1/(3n+3)]$-fraction of the points. Then there is an efficient algorithm to compute $\hat{f}$ everywhere.

Given input x, choose $y \in \mathbb{F}^n$ uniformly at random. Then define $p(\lambda) = \hat{f}(x + \lambda y)$. Note that p is a univariate polynomial of degree at most n because $\hat{f}$ is affine-linear in each of its n variables.

Therefore, we evaluate $p(\lambda)$ for $\lambda = 1, \ldots, n+1$ and interpolate to find $p(0) = \hat{f}(x)$. As before, the probability of error is

$$\Pr[\text{ error }] \ \leq \frac{n+1}{3(n+1)} = 1/3.$$

A coding-theoretic interpretation of this procedure is that we have encoded f using an error-correcting code $\hat{f}$ such that any symbol of $\hat{f}$ can be decoded with a small number of (possibly correlated) queries to $\hat{f}$. To obtain better results, we shall require list-decoding.

If f is a 2^n bit string, we shall encode it as a $2^{n'}$ bit string f'. Suppose that a small circuit C can approximate f' for a $(1/2+\varepsilon)$-fraction of inputs. We shall use a very efficient random self-reducible list-decoder to output a small number of possibilities for f' which we can check quickly.

3 The Hardcore Bit Theorem

We say that a bijection $f : \{0,1\}^n \to \{0,1\}^n$ is a *one-way permutation* if $f \in \mathsf{FP}$, the set of functions decidable in deterministic polynomial time, and if f^{-1} is hard on average.

We say that a predicate $b : \{0,1\}^n \to \{0,1\}$ is a *hardcore predicate* if, given $f(x)$, it is still hard to determine $b(x)$. The content of the Hardcore Bit Theorem is that the predicate $b(x,r) = x \cdot r \pmod 2$ is a hardcore predicate for the function $f'(x,r) = f(x) \circ r$ so long as f is a one-way permutation.

It is worth noting that one-way functions imply that $\mathsf{P} \subsetneq \mathsf{NP}$. It is also true that if f is one-way and b is hardcore then $x \mapsto f(x) \circ b(x)$ is a pseudorandom generator.

Before we prove the Hardcore Bit Theorem in full generality, we shall examine an easier case which illustrates some of the intuition behind the result.

Let f be a one-way function and $b(x,r) = x \cdot r \pmod 2$. Fix an input x. Suppose that we had an algorithm $A(f(x), r)$ which could predict $b(x,r)$ with probability exceeding $3/4 + \varepsilon$. We shall demonstrate how to compute x in contradiction of the fact that f is one-way.

Choose $r \in \{0,1\}^n$ uniformly at random. We shall guess that the ith bit of x is given by

$$x_i = A(f(x), r) \oplus A(f(x), r \oplus e_i),$$

where e_i is the ith basis vector. It is clear that we obtain the correct bit value whenever both of the righthand terms are correct. So we estimate

$$\Pr[\text{ error }] \ \le 2(1/4 - \varepsilon) = 1/2 - 2\varepsilon.$$

Therefore, we can determine each bit of x with probability exceeding one-half. By repeating many times and taking a majority vote, we may determine the whole string with high probability.

Now, we are prepared to prove our main result.

Theorem 3.1 (Goldreich and Levin) *Let f be a one-way permutation on n bits, and let A be an algorithm which predicts $b(x,r) = x \cdot r \pmod 2$ from the quantity $f(x) \circ r$ with probability exceeding $1/2 + \varepsilon$. Then there is an algorithm I which can predict x from $f(x)$ with probability exceeding $\varepsilon/2$ and which runs in time $\mathrm{O}\left((n^3/\varepsilon^4)\mathrm{Time}(A)\right)$.*

Proof. Note that it suffices to output a small list L such that the desired x is in L. This is because f is easy so we can compute $f(y)$ for all $y \in L$ until we find y such that $f(y) = f(x)$, in which case $y = x$. This is like list decoding.

Let ε_x be the probability over r and coin tosses of A that A produces the correct result for a given x minus one-half, i.e.

$$\varepsilon_x = \Pr[\ A(f(x), r) = b(x,r)] - 1/2.$$

By hypothesis, $E(\varepsilon_x) \ge \varepsilon$. It follows using Markov-like reasoning that $\Pr_x[\varepsilon_x \ge \varepsilon/2] \ge \varepsilon/2$. We will invert $f(x)$ for those x with $\varepsilon_x \ge \varepsilon/2$.

Given A which predicts $x \cdot r$, we wish to compute x. The Hadamard encoding of a string x is

$$\mathrm{Had}(x) = \langle x \cdot r \rangle_{r \in \{0,1\}^n}.$$

We know that $A(f(x), r)$ agrees with $x \cdot r$ in at least $1/2 + \varepsilon_x \ge (1+\varepsilon)/2$ positions. We want to find a small number of candidates for x while probing $\mathrm{Had}(x)$ in $\mathrm{poly}(n)$ positions. In other words, we seek an efficient list decoding of the Hadamard code.

Fix x with $\varepsilon_x \geq \varepsilon/2$. If we knew both $b(x,r)$ and $b(x, r \oplus e_i)$ with high probability, we could compute $x_i = b(x,r) \oplus b(x, r \oplus e_i)$ with high probability. And in fact, if we know one of them for sure, it suffices to have a properly biased guess for the other.

To that end, fix $r_1, \ldots, r_k$ independently at random from $\{0,1\}^n$ where $k = \mathrm{O}(\log n/\varepsilon)$. Then, we examine each of the 2^k possibilities for the collection of bits $\{b(x, r_i)\}$ one at a time. One of these possibilities is correct, and the analysis that follows assumes we have the correct possibility. Now, we define $r_{\mathbf{v}} = \sum v_i r_i$ where $\mathbf{v} = (v_1, \ldots, v_k) \in \{0,1\}^k \setminus \{0\}$. By linearity, we deduce that $b(x, r_{\mathbf{v}}) = \sum v_i\, b(x, r_i)$.

Since the random variables $r_{\mathbf{v}}$ are pairwise independent, we have $2^k - 1$ pairwise independent guesses for x_i, the ith bit of x, which we obtain by forming

$$b(x, r_{\mathbf{v}}) \oplus A(f(x), r_{\mathbf{v}} \oplus e_i).$$

Now, we can compute the error probability:

$$\begin{aligned}\Pr[x_i = b(x, r_{\mathbf{v}}) \oplus A(f(x), r_{\mathbf{v}} \oplus e_i)]&\\ &= \Pr[A(f(x), r_{\mathbf{v}} \oplus e_i) = b(x, r_{\mathbf{v}} \oplus e_i)]\\ &\geq \frac{1+\varepsilon}{2}.\end{aligned}$$

So the guesses for x_i are biased to be correct. If we take $2^k - 1 = cn/\varepsilon^2$ samples and conduct a majority vote, we have

$$\Pr[\text{ vote is correct }] \geq 1 - 1/(3n).$$

Since we can compute each bit with error $1/(3n)$, the probability that we can predict x exceeds $1 - n/(3n) = 2/3$.

And the algorithm executes in time $n\, 2^{2k}\, \mathrm{Time}(A)$, which is on the order of $(n^3/\varepsilon^4)\, \mathrm{Time}(A)$.

$\square$