6.842 Randomness and Computation September 20, 2017

Lecture 5

Lecturer: Ronitt Rubinfeld *Scribe: Tom Kolokotrones*

1 Review (Pairwise Independence and Derandomization)

As we discussed last time, we can generate pairwise independent bits using pairwise independent hash functions. Given a prime q, we can define a family of pairwise independent hash functions parameterized by $a, b \in \mathbb{Z}_q$: $\mathcal{H} = \{h_{a,b} : \mathbb{Z}_q \to \mathbb{Z}_q | a, b \in \mathbb{Z}_q, h_{a,b}(x) = ax + b \mod q\}$.

What does it mean to have pairwise independent bits? Sometimes, we talk about pairwise independent bits and sometimes we talk about pairwise independent functions. $\mathcal{H}$ is a subset of the class of all functions $\mathbb{Z}_q \to \mathbb{Z}_q$, which we will denote by $\mathcal{F}$. When we say we are picking a random function we mean that we are randomly choosing a function from $\mathcal{F}$. Since a function $f : \mathbb{Z}_q \to \mathbb{Z}_q$ is uniquely specified by which of the q elements of Z_q to which it maps each of the q elements of Z_q; each function can be thought of as choosing among q options q times, so there are exactly q^q functions in $\mathcal{F}$. On the other hand, since any element of $\mathcal{H}$ is parameterized by $a, b \in \mathbb{Z}_q$, there are q^2 such functions, so $\mathcal{H}$ is much smaller than $\mathcal{F}$. Since q will often be large for us, it is important that the size of $\mathcal{H}$ is only polynomial in q, as opposed to $\mathcal{F}$, which is not.

For many purposes, picking a random function from $\mathcal{H} \subset \mathcal{F}$, instead of from $\mathcal{F}$, will be random enough, since we will often only need the random functions to be pairwise independent. If we randomly choose some $h \in \mathcal{H}$, and we have some collection of x's, $x_1, x_2, \ldots$, then, if we look only at $h(x_1)$ and $h(x_2)$, they will appear perfectly random, as if we'd chosen h from $\mathcal{F}$, but, once we also look at $h(x_3)$, the $h(x)$'s no longer look perfectly random, but they may be good enough, in many cases.

We saw last time how we could use pairwise independent bits, which we can get from pairwise random functions. Sometimes it will be easier to talk about the bits being pairwise independent and other times about the functions being pairwise independent. What we saw last time was that, if you have pairwise independent bits, you can run some algorithms and get exactly the same behavior as if you used fully random bits, or at least sufficiently good behavior to get the desired results.

For **MAX - CUT** we got a 2-approximation, using only pairwise independent bits. Since the required number of pairwise independent bits could be generated using small seeds, we could try every possible seed, of which there was only a polynomial number, so we were able to derandomize the algorithm to a deterministic polynomial time algorithm by using *the method of enumeration.*

2 Reducing Error (The Power of Two-Point Based Sampling)

Today, we'll look at a slightly different use of pairwise independence. We will use it to reduce error. This idea comes from an article called "On the Power of Two-Point Based Sampling" by Benny Chor and Oded Goldreich [1]. We'll look at this setting twice. Today, we will derandomize using pairwise independence. In a future lecture, we will show how to use less randomness using random walks and then compare the two methods.

Assume that we are given any **RP** algorithm: any algorithm that tosses coins and has one-sided error. Last time, **MAX - CUT** wasn't in **RP** because it wasn't stated as a decision problem, but think of any algorithm that works with one-sided error.

Recall the definition of **RP**,

Definition 1 *A language L is in* **RP** *iff, there exists some $\mathcal{A}$, which, on an input of length n uses $R(n)$ random bits and runs in $T(n) \subset \text{poly}(n)$ time, such that, for input x and $r \in \{0,1\}^{R(n)}$:*

$$\begin{array}{ll} \text{If } x \in L: & \Pr_r(\mathcal{A}(x,r) = \textbf{ Accept }) \geq \frac{1}{2} \\ \text{If } x \notin L: & \Pr_r(\mathcal{A}(x,r) = \textbf{ Accept }) = 0 \end{array}$$

Note that, since $\mathcal{A}$ runs in $T(n) \subset poly(n)$ time, it can only use at a most polynomial number of random bits, so $R(n)$ can always be taken to be polynomial in n. In particular, we can never use more random bits than the running time of the algorithm ($T(n)$), and, since the bits are independent and identically distributed, we can always select the first $T(n)$ bits, so we can always take $R(n) \leq T(n)$.

Given $\mathcal{A}$, we can determine whether $x \in L$ with one-sided error $\frac{1}{2}$. However, we may want error $< \frac{1}{2}$. How do we reduce the error?

2.1 Using Fully Random Bits

As we saw last time, we can simply run $\mathcal{A}$ multiple times. We define a new algorithm $\mathcal{A}'$ as follows:

Given an algorithm $\mathcal{A}$ and input x:

1. Run $\mathcal{A}$ on x, k times, using fresh random bits $(r_i), i = 1, \ldots, k$ every time.
2. If, for any i, $\mathcal{A}(x, r_i)$ returns **Accept**, then $\mathcal{A}'$ returns **Accept**.
3. If $\mathcal{A}(x, r_i)$ does not return **Accept** for any i, then $\mathcal{A}'$ returns **Reject**.

Then,

$$\begin{array}{ll} \text{If } x \in L: & \Pr_r(\mathcal{A}'(x,r) = \textbf{ Accept }) \geq 1 - (1 - \frac{1}{2})^k = 1 - (\frac{1}{2})^k \\ \text{If } x \notin L: & \Pr_r(\mathcal{A}'(x,r) = \textbf{ Accept }) = 0 \end{array}$$

If $x \notin L$, no random string can ever cause us to accept, no matter how many times we run the algorithm. Since the k runs each use independent random bits, the errors will be mutually independent, so the probability of incorrectly rejecting when $x \in L$ is $(\frac{1}{2})^k$.

Running $\mathcal{A}'$ uses $R'(n) = k \cdot R(n)$ bits of randomness. We would like to improve this. It's not surprising that $\mathcal{A}$ depends on $R(n)$. Since it calls $\mathcal{A}$, we would expect $R(n)$ to be a lower bound on the number of random bits required. It's also not surprising that it depends on k. However, the question that we will ask is: does this dependence have to be multiplicative? Do we really need $k \cdot R(n)$ random bits or can you do it with $O(k + R(n))$?

2.2 Using Pairwise Independent Bits

So far, what we have done is to use complete independence. What we will do now is a type of recycling of random bits. It will feel like we're getting something for nothing since we'll be able to do a lot of stuff with just a little bit of randomness and we will show that that's good enough for our purposes. One of the reasons that this will be good enough for our purposes is that we had to be really unlucky in order to get a false negative, so maybe we didn't actually need complete independence. Maybe some sort of minimal independence would be good enough. We will be able to get the same bound using fewer bits, although potentially more running time. $\mathcal{A}'$ used $k \cdot R(n)$ bits of randomness and had a running time of $k \cdot T(n)$. Now, we'll try a different idea.

We assume that we're given a class of functions $\mathcal{H}$, which is a collection of pairwise independent hash functions. If $h \in \mathcal{H}$, $h : \left[2^{k+2}\right] = 1, 2, \ldots, 2^{k+2} \rightarrow \{0,1\}^{R(n)}$, so it takes a natural number $\leq 2^{k+2}$ and returns a random string.

We want to be able to pick one of these hash functions randomly with $O(k + R(n))$ random bits in $O(poly(k \cdot R(n)))$ time. In order to choose a hash function, we first need to understand how large q is. We know that q must be at least as large as 2^{k+2}, otherwise, for every hash function, multiple inputs will be assigned to the same random sequence. Likewise, since we have $2^{R(n)}$ random sequences, we must have $q \geq 2^{R(n)}$, otherwise some sequences will never be mapped. Therefore, we must have $q \geq \max(2^{k+2}, 2^{R(n)})$ and so, we can take q to be the smallest prime $\geq \max(2^{k+2}, 2^{R(n)})$. Since this can be found in polynomial time, the running time will remain polynomial. Then, since $\mathcal{H}$ is indexed by $a, b \in \mathbb{Z}_q$, we need $O(\log 2^{k+2} + \log 2^{R(n)}) = O(k + R(n))$ random bits [1].

If we're given a hash function. How are we going to use it? The following procedure shows how. Given a collection of pairwise independent hash functions $\mathcal{H}$ and $O(k+R(n))$ random bits, we can define a new algorithm as follows:

Given a collection of pairwise independent hash functions $\mathcal{H}$,

1. Randomly select a hash function h from $\mathcal{H}$.
2. For i in $1, 2, \ldots, 2^{k+2}$
   (a) Compute $\mathcal{A}(x, h(i))$
   (b) If $\mathcal{A}(x, h(i)) =$ **Accept** then return **Accept**
3. Return **Reject** (if $\mathcal{A}(x, h(i))$ doesn't accept for any i).

The algorithm begins by picking a random hash function. This is the only randomness it uses. Each hash function $h \in \mathcal{H}$ maps $[2^{k+2}] \to \{0,1\}^{R(n)}$, so, instead of picking new random strings, we will use h to generate pairwise random strings by applying h to $1, 2, \ldots, 2^{k+2}$. In the first step, we use $h(1)$, in the second step, we use $h(2)$, in the third step, we use $h(3)$, etc. These will be random strings that are big enough to use as input for $\mathcal{A}$.

In the first step, we compute $\mathcal{A}(x, h(1))$. If it accepts, we know that $x \in L$ because $\mathcal{A}$ never accepts if $x \notin L$. If we see **Accept** then we know that it will accept. If we see **Reject**, then we don't know. It could be that $x \notin L$ or it could be that $x \in L$, but we were unlucky. Next, we compute $\mathcal{A}(x, h(2))$. If we see **Accept**, we know that $x \in L$, but if we see **Reject** we may have been unlucky twice and have to continue. We repeat this process 2^{k+2} times.

The interesting case arises when $k << R(n)$. If $k \geq R(n)$, then it is faster to simply enumerate the $2^{R(n)}$ random strings and not use any randomness. To be concrete, we will think of k as being 10, with the goal of reducing our error to $2^{-10} = \frac{1}{1024}$, so we need to use $2^{10+2} = 2^{12} = 4096$ steps. However, we may need many more random bits. For example, we might need 2000 random bits to run $\mathcal{A}$, so we would need to pick strings of length 2000 to use the algorithm, but this will still be polynomial. We can pick a 2000 bit string in order 2000 time, as well as add, multiply, and test primality, all in polynomial time.

We'll now analyze the performance of our algorithm. We need to look at the time complexity and the number of random bits used. The time complexity is pretty bad: it's $2^{k+2} \cdot T(n)$, whereas with our first algorithm, we had $k \cdot T(n)$. However, using the new algorithm, we only needed to pick $a, b \in \mathbb{Z}_q$, which requires $O(2max(k + 2, R(n)) = O(k + R(n))$ bits vs. $k \cdot R(n)$ for our first algorithm. The new algorithm gives us a much worse running time, but uses a lot fewer bits. It uses an additive, instead of multiplicative, number of random bits, but the running time is not good.

We still don't know whether our algorithm works, but it seems better from one point of view; not the running time, but at least the number of random bits. If we're thinking of k as being 10, then $2^{k+2} = 4096$ isn't so bad. One reason you may want to make this trade of running time for random

[1] The preceding discussion is somewhat imprecise. See Vadhan's Pseudorandomness for an approach using finite fields of nonprime cardinality, which allows for a more general construction of pairwise independent hash functions [2].

bits would be that, in a real algorithm, how do you get randomness? Do you call the clock? Every time you call for randomness, it's a big deal. You have to get some information from somebody, or you could download a bunch of random bits, or you could call the clock, but if you call it too often, your bits will be dependent on each other, so that's known not to be the best thing.

Now, let's see what the behavior is. One side is easy, if $x \notin L$, we are never going to find an **Accept**, so we will always output **Reject**, so at least we're handling $x \notin L$ well. What about if $x \in L$? We're going to misclassify it if we never saw an **Accept** in every single one of the 2^{k+2} times we went through it. Let,

$$\sigma(i) = \begin{cases} 0 & \text{if } \mathcal{A}(x, r_i) = \textbf{Reject} \\ 1 & \text{otherwise } (\mathcal{A} \text{ is correct}) \end{cases}, \qquad Y = \sum_{i=1}^{2^{k+2}} \sigma(i)$$

If $Y = 0$, that's exactly when we do the wrong thing, but if $Y > 0$, we're going to do the right thing. We want to see what the probability is that $Y > 0$, so we do what we always do when we try to figure out the probability that some event happened, we figure out the the expected value of Y and then figure out the probability of deviating from that expected value, and show that it's not very likely that we're going to deviate enough. We'll now compute the expected value of $\frac{Y}{2^{k+2}}$, the expected average value of Y.

$$\begin{aligned} \mathrm{E}\left[\frac{Y}{2^{k+2}}\right] &= \frac{1}{2^{k+2}}\mathrm{E}\left[\sum_{i=1}^{2^{k+2}} \sigma(i)\right] = \frac{1}{2^{k+2}} \sum_{i=1}^{2^{k+2}} \mathrm{E}\left[\sigma(i)\right] = \frac{1}{2^{k+2}} \cdot 2^{k+2}\,\Pr(\sigma(i) = 1) \\ &= \frac{1}{2^{k+2}} \cdot 2^{k+2}\,\Pr(\mathcal{A}(x, r_i) = \textbf{Accept}) \geq 1 \cdot \frac{1}{2} = \frac{1}{2} \end{aligned}$$

where the second equality is due to the linearity of expectation, the third is due to the fact that all of the expectations are equal, and the inequality is due to the definition of **RP**. The expected average value of Y is at least $\frac{1}{2}$. What is the probability that we'll deviate from it? In order to figure this out, we will use two useful lemmas. We can't use Chernoff Bounds here because our random variables are not completely independent. Instead, we can use Chebychev's Inequality.

Definition 2 Chebychev's Inequality — *If X is a random variable with $\mathrm{E}[X] = \mu$ and finite variance, then $\Pr\left[|X - \mu| \geq \epsilon\right] \leq \frac{\mathrm{Var}[X]}{\epsilon^2}$.*

In order to use Chebychev's Inequality, you do not need the random variables to be independent; it's enough if they're pairwise independent. There are other versions of Chebychev's Inequality that are better for k-wise independence. In order to use Chebychev, we need to know the variance of our X, but, if we don't, we can use something else: The Pairwise Independence Tail Inequality.

Definition 3 Pairwise Tail Inequality — *Given t pairwise independent random variables $X_1, X_2, \ldots, X_t$, with $X_i \in [0, 1]$ for all i, let $X = \frac{1}{t}\sum_{i=1}^{t} X_i$ and $\mu = \mathrm{E}[X]$, then $\Pr[|X - \mu| \geq \epsilon] \leq \frac{1}{t\epsilon^2}$.*

Now we can talk what the probability of rejecting an $x \in L$.

$$\begin{aligned} \Pr(\text{Rejecting } x \in L) &= \Pr\left(\frac{Y}{2^{k+2}} = 0\right) \leq \Pr\left[\left|\frac{Y}{2^{k+2}} - \mu\right| \geq \mu\right] = \Pr\left[\left|\frac{Y}{2^{k+2}} - \mathrm{E}\left[\frac{Y}{2^{k+2}}\right]\right| \geq \mathrm{E}\left[\frac{Y}{2^{k+2}}\right]\right] \\ &\leq \frac{1}{2^{k+2}\mathrm{E}\left[\frac{Y}{2^{k+2}}\right]^2} \leq \frac{1}{2^{k+2}\left(\frac{1}{2}\right)^2} = \frac{1}{2^k} = 2^{-k} \end{aligned}$$

where the first inequality is due to the fact that $\frac{Y}{2^{k+2}} = 0$ implies $\left|\frac{Y}{2^{k+2}} - \mu\right| \geq \mu$, the second is from applying the Independence Tail Bound, and the last uses $\mathrm{E}\left[\frac{Y}{2^{k+2}}\right] \geq \frac{1}{2}$, from above. Note that, in the above, $\epsilon = \mu = \mathrm{E}\left[\frac{Y}{2^{k+2}}\right], t = 2^{k+2}$.

We did it! What did we do? We showed that probability of rejecting $x \in L$ is 2^{-k}. Our run time wasn't so great, but our probability of incorrectly rejecting was as good as what happened before when we used fresh randomness each time. We used a lot fewer random bits, we used a lot more time, but we got a probability of error that looks the same, so when your random bits are a more valuable resource than your running time, you could do something like this.

This has been our second use of pairwise independence. We did something really weird: we cut down the number of bits we used. We exploded the time, but we did cut down the number of random bits, so that's kind of interesting. Last time, we cut down the random bits so much that we could just derandomize the algorithm. That was for a specific algorithm, and we knew how it ran. Here we know nothing about the algorithm. We did nothing with the bits in r. We just said, "we don't need fresh random bits, why don't we recycle?"

3 Interactive Proofs (IP)

Next, we'll use pairwise independence in yet another way, in order to get a complexity theoretic result for interactive proofs. In order to do that, we need to talk about what interactive proofs are. I'm going to show you a famous interactive proof that you may have already seen before, but if you haven't you should see it. A problem that we won't get to this time is a really cool result about whether the coins need to be private or public, which is a complexity theoretic result, but is such a beautiful idea, that I like to teach it anyway because I've used it in my research for other things.

I want to talk about Interactive Proofs, which are a really cool idea of Shafi Goldwasser, Silvio Micali, and Charles Rackoff (the first two are from here and got a Turing Award in 2012). It's a beautiful notion that is very important in cryptography.

NP is all the decision problems for which a yes answer can be verified in polynomial time. There is some witness that you can write down and you can take that witness as input and say, "oh yeah, that's a good witness," and the verification of the witness can be done in polynomial time. For example, if you give me a graph and a Hamiltonian path, I can check it in polynomial time. That's **NP**. For every problem in **NP**, there's some witness that a polynomial time verifier can check and say, "yeah, that's a good witness, I believe you that x is in the language." We can say nothing about x not being in the language. If x is not in the language, I don't claim that you have a witness, that would be **coNP**, but we're talking about **NP**: just things in the language have a short proof.

IP, Interactive Proof. This is a new class. **IP** is a generalization of **NP**. It's a broader class of languages, we think. We don't actually know, but we think. We don't actually know anything (in complexity) so in that scheme of things, it's actually not surprising that we don't know this. It's a new model. Here, starting with the notion of short proofs from **NP**, we're going to generalize to short interactive proofs. In other words, there's some conversation that we can have, where you can convince me that something is true. Let me give you the best example that I know.

There was once a thing called the Pepsi Challenge. There was a time when Pepsi was trying to convince people that they preferred Pepsi over Coke and there were people who swore that they could tell the difference. Let's say you want to prove to me that you can actually tell the difference between Coke and Pepsi. How would we do this? In secret, I would pour a glass of Pepsi and a glass of Coke and, while you were behind a wall, I would switch them around and then I would give you one and see if you could identify whether I gave you the Pepsi or the Coke. Maybe the first time you would get it right, since you always have a 50-50 chance of getting it right with random guessing, but I'll do this k times. If you get it right after k times, I'm going to start to believe that maybe you do know Pepsi from Coke. This is the idea of an interactive proof.

An interesting example that we're going to talk about is graph nonisomorphism. Graph isomorphism is known to be in **NP** and was recently shown to be in quasipolynomial time, but, until last year, we didn't even know it was in quasipolynomial time and were kind of hoping that this would be one of our candidate hard problems [3]. There are other examples out there that we don't know are in **NP**, but have interactive proofs. For graph isomorphism, we don't need an interactive proof, because we have a proof.

If I want to prove that two graphs are isomorphic, I can give you the relabeling of the node names, so that you can check if they're really isomorphic, but if they're not isomorphic, how would you prove that? Would you go through all possible permutations of relabelings and show that each one doesn't work? That's an awful lot, that's $n!$, so that's not a good thing to do. We don't know of a way, in polynomial time, to solve graph isomorphism (only in quasipolynomial time). In particular, we don't know a short proof for graph nonisomorphism, so we don't know how to put graph isomorphism into **coNP**, but we do know how to give an interactive proof. It will be based on the Pepsi Challenge idea. If you know two graphs aren't isomorphic, if I randomly permute the names of my graph and I send it to you, you'll know which one I sent you.

The reason we're interested in interactive proofs is that we're going to talk about the number of random bits and how the bits are presented: whether the coins used have to be kept secret or whether they can be out there in the public. We'll use the idea of pairwise independent hashing to get this protocol to show something very exciting.

References

[1] Chor, Benny and Goldreich, Oded. Journal of Complexity 5, 96-106 (1989).

[2] Vadhan, Salil. Pseudorandomness. Foundations and Trends in Theoretical Computer Science Vol. 7, Nos. 1-3 (2011) 1-336.

[3] Babai, Laszlo. Graph Isomorphism in Quasipolynomial Time. arXiv:1512.03547. December 11, 2015.