CS640 **Computational Complexity**

Derandomization: The Deathly Hallows

Instructor: Manindra Agrawal *Scribe: Ramprasad Saptharishi*

Last class we revisited MA protocols. And by putting certain restrictions on the size of the proof, number of random bits used by verifier, and the number of probes into the proof, we came up with the definition of MIP. We stated the following theorem:

Theorem 1. $\mathsf{MIP} = \mathsf{NEXP}$

In this case we had exponential sized proofs and polynomial many probes and random bits. We wanted to see if a scale-down can tell us anything about NP. Then we stated the PCP theorem:

Theorem 2. $\mathsf{NP} = \mathsf{PCP}(O(\log n), O(1))$.

Infact the constant number of probes can be made to 3. The philosophical interpretation of this is amazing: some gives you a polynomial sized proof of the statement that $x \in L$ and all you need to do is read 3 bits of this proof to verify whether the proof is right or not with good probability.

This came as a *huge* surprise to complexity theorists and is considered one of the crowning achievements in this field. If one needs a picture of why this could be true, look at the proof as an error-correcting code. So the idea is that, if the proof was incorrect, then it differs from a right proof (if any) at some places. Encode it with an error correcting code. Then, even if there was a single place where it differed from the actual proof, it would now propogate to a lot of places in the encoded word. So finding an error there is a lot easier.

The original proof is quite complex and involves a builds up on a lot of earlier results. A much simpler proof of the PCP theorem was given by Irit Dinur in 2005. We shall be doing her proof in this course.

1 The Deathly Hallows

We shall make a digression and look at randomization again and certain combinatorial objects that can help reduce (or remove) randomness from algorithms.

We shall discuss 3 combinatorial objects:

1. Something that squeezes out “nearly uniform” bits from “partially random” source.
2. Something that reduces our dependence on random bits used in algorithms
3. Something that eliminates randomness all together

One may wonder why we need the second object if we can eliminate the randomness all together. The fact is that we know how to make create the second object but not the third.

1.1 Extractors

This is our first object - to extract “nearly uniform” bits from a “partially random” source. We first need to formally define what we mean by partially random or nearly random distributions.

1.1.1 Some Parameters on Distributions

Let X be a probabilistic distribution over $\{0,1\}^n$. If X were the uniform distribution, then for every string $s \in \{0,1\}^n$, $\Pr[X = s] = 2^{-n}$. But if the distribution was skewed then this need not be true.

We need some way of characterizing how skew distributions are. Suppose the distribution was such that some string like $00 \cdots 0$ appeared with probability $1/2$, then the distribution is really skewed; most of the time you would sample this string.

Hence, one way to characterize the skewness is to look at the item that gets sampled with highest probability; this is certainly atleast 2^{-n}. If this wasn't too large, then we intuitively see that it's not too skewed. This is captured by the definition of *min-entropy.*

Definition 1. *A distribution X over $\{0,1\}^n$ is said to have min-entropy k if*

$$\max_{s \in \{0,1\}n} \Pr[X = s] \leq \frac{1}{2^k}$$

Note that min-entropy of any distribution over n-bit strings is between 0 and n. If the min-entropy was n then we already have a uniform distribution.

Intuitively, if min-entropy was k, then the distribution has essentially only k truly random bits. Interpret this over the following distribution over

n bit strings. Define the probability as 2^{-k} for all strings that have the last $n-k$ bits as zeroes. In essence, this is just a uniform distribution over the first k bits and padding them up with zeroes.

Thus, any distribution with min-entropy has k truly random bits inside. We are interested in extracting them somehow.

Next is the notion of "nearly uniform." What does it mean to say a distribution is almost random? That is captured by the following distance measure between two distributions.

Definition 2. *For any two distributions X, Y over $\{0,1\}^n$, define the distance between them as*

$$\delta(X,Y) = \frac{1}{2}\sum_{s\in\{0,1\}^n} |\Pr[X=s] - \Pr[Y=s]|$$

If X was indeed equal to Y then the distance is zero. And since

$$\begin{aligned}\frac{1}{2}\sum_s |\Pr[X=s] - \Pr[Y=s]| &\leq \frac{1}{2}\sum_s \Pr[X=s] + \frac{1}{2}\sum_s Pr[Y=s]\\ &= \frac{1}{2}+\frac{1}{2} = 1\end{aligned}$$

the distance is always a number between 0 and 1. The distance is equal to 1 if the two distributions are orthogonal[1].

And with this definition we can formally define the notion of "nearly random".

Definition 3. *A distribution X is said to be ϵ close to uniform if $\delta(X, U_n) \leq \epsilon$ where U_n is the uniform distribution over n-bit strings.*

Now if you look back at all the randomized algorithms we have done, they assume truly random bits. It can be seen that instead of that, if we were to provide them with random bits that are ϵ close to uniform, then the error probability just goes from $1/3$ to $1/3+\epsilon$.

With these, we can now define our extractors formally.

Definition 4. *A (k,ϵ) extractor is a polynomial time computable function $E : \{0,1\}^n \times \{0,1\}^t \longrightarrow \{0,1\}^m$ such that for any distribution X over $\{0,1\}^n$ of min-entropy at least k,*

$$\delta\left(E(X,U_t), U_m\right) \leq \epsilon$$

[1]for any string s, either $\Pr[X=s]=0$ or $\Pr[Y=s]=0$

So E is an extractor that takes a distribution of reasonable min-entropy, and a random *seed* of t purely random bits and *extracts* out the randomness from X. This is meaningless if t was large. For instance if t was as large as m, then we can just ignore the first parameter of E and just output the uniform bits.

However, the power of this is seen when t is much much smaller than n or m. We shall prove the following theorem later.

Theorem 3. *There exists an (n^δ, ϵ) extractor E where*

$$E : \{0,1\}^n \times \{0,1\}^{O(\log n)} \longrightarrow \{0,1\}^{n^{\delta'}} \qquad \delta' \le \delta$$

This in essence means that you need your random seed as small as just logarithmic in the m and n. With just that many seeds, you can extract almost all the random bits from X.

It can be shown that the random seed is essential and that to do any reasonable extraction, you need $O(\log n)$ random bits as a seed. Of course, in the next iteration one could use the almost uniform random bits as a seed. But we need $O(\log n)$ random bits to start with.

In a sense, the seed is a catalyst to the reaction, you need it to start the process; without that we won't be able to extract any randomness.

1.2 Expanders

This is the object that will help us reduce the number of random bits used in randomized algorithms. This has a totally different definition compared to extractors; these are graphs.

Definition 5. *A d-regular graph G on n vertices is called an (n, d, ϵ)-vertex expander if for every subset S of size less than $n/2$, the number of edges going out of S is atleast $\epsilon d|S|$. That is, $E(S, V \setminus S) \ge \epsilon d|S|$.*

Since G is a d-regular graph, the number of vertices incident on S is $d|S|$. The definition of expander says that an ϵ fraction of those edges actually go out of S.

1.2.1 Boosting Success Probability using Expanders

Let us take some randomized algorithm we did in class. To see the power of expanders better, let us take an algorithm whose error is one-sided. We saw numerous algorithms where if x was in the language, then the algorithm accepts with probability 1. The error was only when $x \notin L$, we could incorrectly accept x with some small probability. Let us say this error was $1/3$ to begin with. We want to boost this error.

Our approach was to repeat this algorithm k times and therefore the error probability would come down to $(1/3)^k$. What is the number of random bits used in the process? Suppose a single round used m purely random bits. Then this method of boosting would use km random bits totally.

We shall see how this can be reduced substantially using expander graphs. Look at all the possible random strings you could get. The algorithm uses m random bits, the number of possible random strings is $n = 2^m$. Of these strings, some of them lead the algorithm to the right answer, and some of them don't. Let S be the set of bad strings, those that lead the algorithm to the wrong answer. Since the error is bounded by $1/3$, it follows that $|S| < n/3 < (1/2)n$.

Consider a (n, d, ϵ) expander where each vertex encodes a sequence of m bits. Note that d, ϵ are constants. Now S, the set of bad vertices, is a subset whose size is less than $n/2$ and therefore has the expansion property. Pick a random vertex v. If this vertex was outside S, then we already have a witness. The bad case when we get a vertex inside S. At this point, pick one of the d neighbours of S at random and move to vertex. Note that, by the expansion properties of the graph, an ϵ fraction of the edges go out. Therefore, with prob at most $(1-\epsilon)$ you will remain in S. Now repeat this picking a random neighbour and moving to that for say k times. Then the probability that this entire walk stayed inside S is about $(1-\epsilon)^k$. In this way, you boost the error probability to again inverse exponential just like the naive independent trials method.

But what is the total number of random bits used? The first vertex to be picked needed m random bits. After that, to pick a neighbour at random, we just need $\log d$ random bits. And since d is a constant, this is just contant number of random bits. Therefore, the total number of random bits used is $m + O(k)$ which is way better than mk.

Notice that there is a small catch here. How do we construct this expander? The problem is that the size is *huge*, 2^m vertices. How do we even construct such graphs? Notice that you really don't need to store the entire graph (since that would be exponential space and would destroy all hopes of efficient computation). All we need is, given a vertex v, what are the neighbours of v?

This is accomplished by maps called *rotation maps*. These are maps that take u and an integer $i \leq d$ as input and returns the i-th neighbour v of u and a j such that u is the j-th neighbour of i. And fortunately, expanders can be constructed such that this map is polynomial time computable; we shall see this soon.

And thus, we can reduce the number of random bits used in the boosting.

1.3 Pseudo-random Generators

This is the third object; the one that completely eliminates randomness. Let us first the formal definition of it and then look at the intuition.

Definition 6. *Let $f : \{0,1\}^\star \longrightarrow \{0,1\}^\star$ be a function that satisfies the following properties:*

- *$f(x)$ is computable in $2^{O(|X|)}$ time.*
- *There exists a constant c such that on any string x of size $c \log n$, $f(x)$ is a string of size less than n. In other words, f stretches strings of $O(\log n)$ bits to n bits.*
- *For every circuit C of size n with n inputs,*

$$\left| \Pr_{x \in \{0,1\}^n}[C(x) = 1] - \Pr_{y \in \{0,1\}^{c \log n}}[C(f(y)) = 1] \right| \leq \frac{1}{n}$$

Then f is called an optimal pseudo-random generator against linear sized circuits.

First let us explain the adjectives used here. It is optimal this pseudo-random generator is stretchings of logarithmic size to n bits. It can be shown that there does not exist any pseudo-random generator that stretches some function less than $\log n$ to n bits. And hence, in that sense it is optimal.

It is called a pseudo-random because the output of f is strings of $\{0,1\}^n$ but they aren't really random but appear random to the circuit C. To understand this better, think of C as a randomized algorithm. Suppose its input was fixed and C is just look for the random coin tosses so that it can give the answer.

What we know is that if x is a string in the language, then $\Pr_{x \in \{0,1\}^n}[C(x) = 1] > 2/3$ is atleast and if it was not in the language this is less than $1/3$. Suppose instead of x, we pick up a random string y of size $c \log n$ and give $f(y)$ to C. Then by the third property, we know that $\Pr_{y \in \{0,1\}^{c \log n}}[C(f(y)) = 1]$ is just about $1/n$ away from $1/3$ or $2/3$ and therefore is still good enough. In essense, instead of giving C really random bits, f provides it with bits that aren't really random but still manages to make C work properly. That is what is meant by pseudo-random generators *against* linear sized circuits.

Another way to look at it is think if C as a distinguisher circuit - tries to distinguish between random and pseudo-random bits provided as input. The third property tells us that C cannot really distinguish between them since the output of C on real random bits and outputs of f isn't too far away from each other. Therefore, the function f is able to *fool* C into thinking that it is actually providing it with truly random bits.

Now notice that f is stretching strings as small as $c \log n$ bits to n bits. Therefore, we can just run over every string $y \in \{0,1\}^{c \log n}$ and find the fraction of points where $C(f(y)) = 1$. We are guarenteed that this is very close to $1/3$ or $2/3$ and therefore we will know for sure how C would behave if provided with truly random bits.

Thus, by running over every possible y, which can be done in polynomial time since f is computable in $2^{O(|x|)}$ time, we can completely remove randomness from the algorithm C. Summarizing this as a theorem:

Theorem 4. *If there exists an optimal, pseudo-random generator against linear sized circuits, then* $\mathsf{BPP} = \mathsf{P}$. $\square$

Unfortunately, we do not know of any way to find such PRGs at the moment. But there are strong evidences to believe that such PRGs exist. And hence, people believe that $\mathsf{BPP} = \mathsf{P}$. However finding a PRG of this kind doesn't mean we can derandomize classes like AM. To remove randomness from higher classes, we need more powerful PRGs.

We shall get back to this after our discussion on expanders and the PCP theorem. We will see that PRGs are strongly connected to extractors.