A Theorist's Toolkit **(CMU 18-859T, Fall 2013)**

Lecture 11: Derandomization

October 14, 2013

Lecturer: Ryan O'Donnell *Scribe: Linus Hamilton*

1 The Benefits Of Being Random

There are many problems which we only know how to solve efficiently by being random. For example, say we want to know whether a polynomial expression like $P(\bar{x}) = (x_1+x_3-2x_2)^4-(x_3+x_2)^4-(x_1+x_2)^4$ is identically zero. If we don't mind a small chance of being wrong, we can just choose random values for x_1, x_2, and x_3, and check whether $P(\bar{x}) = 0$. Usually these calculations are performed over a large field $\mathbb{F}_q$; then the Schwartz-Zippel Lemma from Lecture 9 guarantees that we will get the right answer with a very large probability.

This notion of "usually getting it right" is formalized with the complexity class BPP:

Definition. BPP is the class of all problems solvable in polynomial time with a randomized algorithm, such that no matter what the input is, the algorithm outputs the correct answer with probability $\geq \frac{2}{3}$.

Remark. $\frac{2}{3}$ can be replaced with any constant $\frac{1}{2} < c < 1$, or even any function between $\frac{1}{2}+k^{-c}$ and $1-2^{k^c}$, where k is the length of input. This is because we can always reduce an algorithm's error by running it over and over again.

The argument above shows that Polynomial Identity Testing - the problem of determining whether P is the zero polynomial - is in BPP. However, it is open whether this problem is in P! (Just expanding the polynomial doesn't work, because there might be exponentially many terms.)

This lecture is all about derandomization - the process of taking an efficient randomized algorithm, and (hopefully) transforming it into an efficient deterministic algorithm.

2 Pseudo-Random Generators

Here's an idea. Instead of using random numbers in our algorithm, why not use a sequence of pseudorandom numbers instead?

Definition. A pseudorandom generator (PRG) is a function $G : \{0,1\}^\ell \to \{0,1\}^n$. ℓ is called G's seed length.

Of course, we don't want pseudorandomness to alter our results. If our program returns "1" most of the time with random bits, we also want it to return "1" most of the time with pseudorandom bits. This is formalized by the notion of "ϵ-fooling."

Definition. Let $\mathcal{C}$ be a class of functions $f : \{0,1\}^n \to \{0,1\}$. A PRG G "ϵ-fools" $\mathcal{C}$ if for every $f \in \mathcal{C}$,

$$|\mathop{\mathbf{Pr}}_{s\in\{0,1\}^\ell}[f(G(s)) = 1] - \mathop{\mathbf{Pr}}_{r\in\{0,1\}^n}[f(r) = 1]| < \epsilon.$$

Why is ϵ-fooling useful? Well, suppose we have a randomized algorithm $\mathcal{A}$ that takes input x along with n random bits, and produces an answer in some polynomial - say, $O(n^{10})$ - time. Further suppose that $\mathcal{A}$ always returns the right answer with probability at least 0.99. (As we saw in the definition of BPP, the constant here doesn't matter.)

Let $\mathcal{A}(x, r)$ denote the bit that $\mathcal{A}$ returns when given input x and random bits r. Then for any fixed x, the function $r \mapsto A(x, r)$ is computable by a $O(n^{10}\text{polylog}(n))$-sized circuit (see the Computational Models lecture).

Let $\mathcal{C}$ be the class of functions that are computable by a $O(n^{11})$-sized circuit. Then $\mathcal{C}$ includes all functions $r \mapsto \mathcal{A}(x, r)$.

Now, here is the key idea. Suppose are lucky enough to find a PRG G with seed length ℓ that 0.4-fools $\mathcal{C}$. Then, by the definition of ϵ-fooling,

$$|\mathop{\mathbf{Pr}}_{s\in\{0,1\}^\ell}[\mathcal{A}(x, (G(s)) = 1] - \mathop{\mathbf{Pr}}_{r\in\{0,1\}^n}[\mathcal{A}(x, r) = 1]| < 0.4$$

$$\implies \mathop{\mathbf{Pr}}_{s\in\{0,1\}^\ell}[\mathcal{A}(x, G(s)) \text{ gives the wrong answer}] < 0.01 + 0.4 = 0.41.$$

Instead of running $\mathcal{A}$ on n random bits, we need only run it on $G([\ell\text{random bits}])$! This drastically cuts down on the amount of random bits needed.

Now, you may say, "So what?" Unless we are running short of random-bit-generating lava lamps, this doesn't seem like a big improvement. But here's the key: if $\ell = O(\log n)$, then we can run $\mathcal{A}$ with *every* seed, one at a time, in polynomial time. Then, by counting which result ("yes" or "no") we get more often, we can exactly solve the original problem, with no chance of being wrong.

This generalizes to our first big derandomization result:

Claim. If, for every c, there is a PRG G with seed length $O(\log n)$ that 0.4-fools the class of functions computable by a circuit in $O(n^c)$ time, then BPP = P.

Proof. Use the logic above, but replace 11 by c. □

In their quest to prove BPP = P, complexity theorists have strengthened the above result in a few different ways, usually by making the hypothesis more and more plausible. In the next section, we'll see an outline of one way people might end up proving BPP = P.

3 Does BPP = P? Hardness vs. Randomness

3.1 Intuition

In this section, I want to give you intuition about a link between hardness and randomless. Specifically, a statement like "if we can come up with hard problems, then BPP = P."

In the last section, we saw that we can derandomize BPP if we have a family of PRGs $\mathcal{G}$ which ϵ-fools all polynomial-size circuits. The key intuition is that these two statements are the same:

- "$\mathcal{G}$ ϵ-fools all polynomial-size circuits."
- "The problem 'Does this output come from $\mathcal{G}$ or a truly random generator?' is too hard to solve in polynomial time."

Let's suppose we believe that the famous hash functions SHA-1, SHA-2, ..., and SHA-n are real one-way functions, i.e., they are hard to invert. Then we can define a PRG G_{SHA} which maps x to the concatenation (SHA-1(x), SHA-2(x), ..., SHA-n(x)). To tell the difference between G_{SHA} and a true random number generator, you'd have to solve the problem "Is there an x which generates this sequence of results?" If SHA is a good one-way function, then this is a hard problem, maybe even NP-complete. If you can prove that this problem is too hard for polynomial-size circuits, then you'd have proven $\mathsf{P} = \mathsf{BPP}$.

(Oh, and also $\mathsf{P} \neq \mathsf{NP}$, but that's another matter...)

After all this buildup, you probably want a theorem...

Theorem. *(Impagliazzo-Wigderson) Suppose that for all m there exists a function $h_m : \{0,1\}^m \to \{0,1\}$ that is computable in time 2^{100m} by a Turing Machine, but not by $2^{.001m}$-size circuits. Then* $\mathsf{P} = \mathsf{BPP}$.

The idea of the proof is the same as the SHA-sequence above, but with some delicacies. For example, each bit of the proof's PRG's output depends not on the whole input, but on a certain m-size subset of the input bits. The subsets overlap in a special way that ensures there are no shortcuts - that someone actually has to solve h_m to be able to tell the PRG's output apart from a random stream.

Here's another theorem:

Theorem. *(Nisan '92) There is a PRG G which ϵ-fools the class of randomized algorithms using n random bits and $S(n) \geq \log n$ space, where:*

- $\epsilon = 2^{-S}$
- *The seed length is $O(S(n) \log n)$*
- *G is computable in $O($seed length$)$ space.*

Starred Exercise: Using this theorem, prove $\mathsf{BPPSPACE} = \mathsf{PSPACE}$. (BPPSPACE is the same as BPP, only you get polynomial space instead of polynomial time.)

Remark. Many if not most complexity theorists think that $\mathsf{BPP} = \mathsf{P}$. And we know $\mathsf{BPP} \subset \mathsf{P\backslash poly}$. (Proof: pretend the advice string is output from a really good PRG.) But we can't even prove that $\mathsf{BPP} \neq \mathsf{NEXP}$.

4 Something Completely Different: Conditional Expectations

PRGs gave us an extremely general method to try to derandomize *all* algorithms. But we can derandomize some specific algorithms without going through all the hassle. One useful method is the Method of Conditional Expectations. The best way to explain it is

by example. Consider the MAX-CUT problem: given a graph $G = (V, E)$, we want to find a subset of vertices $S \subseteq V$ that maximizes the number of edges that connect S to $G \backslash S$.

Here's a randomized algorithm: pick a completely random subset S. That is, put every vertex in S with probability $\frac{1}{2}$ independently. Assuming there are no self-loops, every edge has a $\frac{1}{2}$ chance of bridging S and $G \backslash S$, so **E[cut size]** $= \frac{1}{2}|E|$.

We'd be hard-pressed to deterministically obtain a cut of size at least $\frac{1}{2}|E|$ using the PRG techniques above, because so many questions about whether or not PRGs exist are still open. However, there *is* a deterministic algorithm that always outputs a subset S with cut size at least $\frac{1}{2}|E|$. Here it is, in pseudocode:

- Initialize an array A which can label each vertex as "in", "out", or "undecided". Initially, all of the vertices are "undecided".
- Define the function ExpectedScore(A) as follows. Let S_A be a random subset of vertices, obtained by taking A, changing all "undecided"s to "in" or "out" independently with probability $\frac{1}{2}$, and then setting S_A to be the set of all vertices which are "in". Then ExpectedScore(A) = **E[cut size of** S_A**]**.
- For v in vertices:
  - By the end of this loop, we will have decided whether $v \in S$ or $v \notin S$.
  - Let A_{in} equal A, but with $A[v]$ set to "in".
  - Let A_{out} equal A, but with $A[v]$ set to "out".
  - Calculate ExpectedScore(A_{in}) and ExpectedScore(A_{out}).
  - If ExpectedScore(A_{in}) is bigger, set $A[v]$ ="in". Else, set $A[v]$ ="out".
- Finally, return S, the set of vertices which are "in".

Why does this work? Well, by the definition of ExpectedScore, we always have ExpectedScore(A) $= \frac{1}{2}$ExpectedScore(A_{in}) $+ \frac{1}{2}$ExpectedScore(A_{out}). Therefore, since we always follow the path whose expected score is bigger, ExpectedScore(A) never decreases during the course of this algorithm. Initially, ExpectedScore(A) $= \frac{1}{2}|E|$, so we always return a set whose cut size is at least $\frac{1}{2}|E|$, as desired.

The key for this problem is that ExpectedScore is easy to compute. At every step, every edge is either known to cut S, known to not cut S, or will cut S with probability $\frac{1}{2}$. So we can compute ExpectedScore in $O(\#$ of edges), and thus the algorithm runs in polynomial time.

In general, the Method of Conditional Probability works whenever it's easy to calculate "how well we're doing so far", i.e., the expected score if we make the rest of the decisions randomly.

5 Back to PRGs: Special Kinds of PRGs

Sometimes, we don't need the full power of an ϵ-fooling PRG. We would rather have an object that isn't quite so strong, but that we actually know exists. Here are two such cases.

5.1 k-wise Independent PRGs

Definition. A PRG $G : \{0,1\}^\ell \to \{0,1\}^n$ is *k-wise independent* if:

- For all indices $1 \le i \le n$, the i^{th} bit of G's output is 1 with probability exactly $\frac{1}{2}$.
- For all sets of k distinct indices $1 \le i_1, i_2, ..., i_k \le n$, the i_1^{th} bit of $G(s)$, i_2^{th} bit of $G(s)$,..., i_k^{th} bit of $G(s)$ are all independent. That is, each of the 2^k possible sequences of values has a $\frac{1}{2^k}$ chance of appearing.

Remark. This definition generalizes easily to alphabets other than $\{0,1\}$. Just replace "2^k" with "$\lfloor$size of the alphabet$\rfloor^k$".

Theorem. *(ABI '85) For all $k \le n$, and all prime powers q, there exists a PRG $G : \mathbb{F}_q^\ell \to \mathbb{F}_q^n$ which is computable in polynomial time, k-wise independent, and has seed length $\lfloor \frac{k}{2} \rfloor log_q n + O(1)$.*

This gives an alternate deterministic algorithm for getting $\frac{1}{2}|E|$ in MAX-CUT: use the completely random algorithm, but with a 2-wise independent PRG. Every edge still cuts S with probability $\frac{1}{2}$, since its endpoints are still in S independently with probability $\frac{1}{2}$. So, the expected cut size is still $\frac{1}{2}|E|$. But now, the seed length is only $O(\log n)$. So we can check all possible seeds in polynomial time, guaranteeing success.

5.2 ϵ-bias PRGs

Definition. A "ϵ-bias PRG" is one that ϵ-fools the class of functions $f_w : r \mapsto w \cdot r$, where $w \in \mathbb{F}_2^n$.

Theorem. *(NN '93) There is a polynomial-time-computable ϵ-bias PRG with $O(\log \frac{n}{\epsilon})$ seed length.*

It's not as clear how to use this PRG, but the example in the lecture used it to solve a decision problem version of matrix multiplication quickly. Say we're given matrices A, B, and C in $\mathbb{F}_2^{n \times n}$. We want to determine whether $AB = C$.

The deterministic world record for this problem is $O(n^{2.73...})$. But we can solve it using randomness in $O(n^2)$!

Here's how: pick a random vector $v \in \mathbb{F}_2^n$. Compute $A(Bv) - Cv$, and check whether it is the zero vector.

If $AB = C$, then it will always be the zero vector. If $AB \neq C$, then $AB - C \neq 0$, so there is a nonzero column w in $AB - C$. One of the entries in $A(Bv) - Cv$ will be equal to $w \cdot v$, and this entry will be 0 exactly half of the time. So, this algorithm is wrong with probability at most $\frac{1}{2}$. (And of course we can repeat it over and over again to get any desired accuracy.)

All that this random algorithm required is that, for every nonzero vector w, $w \cdot v$ has a nonzero chance of returning 1. Conveniently, this is exactly what an 0.4-bias generator does. So we get a deterministic polynomial-time algorithm, by using a 0.4-bias generator and checking all possible seeds of length $O(\log \frac{n}{\epsilon})$.

Remark: Unfortunately, this deterministic algorithm requires checking $n^{O(1)}$ seeds, so depending on the constant, it may not be better than the $O(n^3)$ alternative.

5.3 PRGs and Coding Theory

Surpisingly, we can make k-wise independent PRGs and ϵ-bias generators using coding theory.

Getting 2-wise Independence With Hadamard Codes

We can use a Hadamard code to create a 2-wise independent PRG. Here's how. Given $x \in \{0,1\}^\ell$, define $G(x) = Hx$, where H is the $(2^\ell - 1)$-by-ℓ Hadamard code's matrix:

$$H = \begin{pmatrix} 0 & \cdots & 0 & 1 \\ 0 & \cdots & 1 & 0 \\ 0 & \cdots & 1 & 1 \\ & \vdots & & \\ 1 & \cdots & 1 & 0 \\ 1 & \cdots & 1 & 1 \end{pmatrix}.$$

Claim. G is 2-wise independent.

Proof. It's pretty clear that every bit of G is 0 or 1 with probability $\frac{1}{2}$ - it's the same as saying that if you flip a nonzero number of coins, the probability that you get an odd number of heads is $\frac{1}{2}$. So if suffices to show that if v and w are two different rows of H, then $v \cdot x$ and $w \cdot x$ are independent as x varies inside $\{0,1\}^\ell$.

There is at least one index i where $v_i \neq w_i$. WLOG $v_i = 0$ and $w_i = 1$. Take a random vector x. Instead of randomly generating x all at once, first randomly generate all the indices except the i^{th}. Now $v \cdot x$ is fixed, but $w \cdot x$ is still 0 or 1 with probability $\frac{1}{2}$. Therefore, $w \cdot x$ and $v \cdot x$ are independent, as desired. □

Getting ϵ-bias Generators With Reed–Solomon Codes

Here is Ryan O'Donnel's explanation, mostly pasted from Piazza.

Suppose $G : \mathbb{F}_2^\ell \to \mathbb{F}_2^n$ is an ϵ-biased generator.

Let M be the $2^\ell \times n$ matrix (over $\mathbb{F}_2$) whose rows are all the outputs of G (in a sense, the "truth table" of G). Now consider Mw for $w \in \mathbb{F}_2^n$. Mw is the vector of outputs $\{G(v) \cdot w | v \in \mathbb{F}_2^\ell\}$. If you go back and read the definition of ϵ-bias generators, you see that G is indeed an ϵ-biased generator iff for all nonzero w the vector/string Mw has relative (i.e., fractional) Hamming weight in the range $[\frac{1}{2} - \frac{\epsilon}{2}, \frac{1}{2} + \frac{\epsilon}{2}]$. I.e., iff M is the generator of a linear code with minimum relative distance at least $\frac{1}{2} - \frac{\epsilon}{2}$ and also (here's the slight twist) maximum relative distance at most $\frac{1}{2} + \frac{\epsilon}{2}$.

To get a generator with good seed length we want M to be a fat matrix; i.e., we want the code to have good rate. More precisely, any $[N, n, (\frac{1}{2} - \frac{\epsilon}{2})N]_2$ code where all codewords also have Hamming weight at most $(\frac{1}{2} + \frac{\epsilon}{2})N$ yields an ϵ-biased PRG stretching $\log N$ bits to n bits.

If you recall from Lecture 10, we saw that we can choose parameters for a Reed–Solomon code so that, when concatenated with the Hadamard code, it is a $[q^2, \epsilon q log q, (\frac{1}{2} - \frac{\epsilon}{2})q^2]_2$ code (for any prime power q). Further more, since all nonzero Hadamard codewords have relative Hamming weight exactly $\frac{1}{2}$, it's easy to see that every codeword of the concatenated code has relative Hamming weight at most $\frac{1}{2}$ (which is even better than the $\frac{1}{2} + \frac{\epsilon}{2}$ we could have tolerated). Thus we get a quite-decent, efficiently computable

ϵ-biased generator this way, the one proposed by Alon–Goldreich–Håstad–Peralta. If we call n its output length and just sloppily use $n = \epsilon q log q \geq \epsilon q$, then the seed length, q^2, is at most $2 \log \frac{n}{\epsilon}$. Actually, I think if you are being super-careful about whether n and ϵ are powers of 2, you might decide that the seed length is upper-bounded by $2 \log \frac{n}{\epsilon} + 4$ or something.

Recommended Reading

Ryan O'Donnell's recommendations:

http://pages.cs.wisc.edu/~dieter/Courses/2013s-CS880/Scribes/lectures.html

http://people.seas.harvard.edu/~salil/pseudorandomness/

Similar notes: http://people.seas.harvard.edu/~salil/cs225/spring09/lecnotes/Chap3.pdf

The Complexity Zoo: https://complexityzoo.uwaterloo.ca/ (Or whatever URL it ends up at next)

"Starred Reading": http://www.wisdom.weizmann.ac.il/~oded/COL/bpp-p.pdf ($\mathsf{BPP} = \mathsf{P}$ iff good PRGs exist)

References

[1] Russell Impagliazzo and Avi Wigderson. P = bpp if e requires exponential circuits: derandomizing the xor lemma. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 220–229, New York, NY, USA, 1997. ACM.

[2] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 213–223, New York, NY, USA, 1990. ACM.

[3] Noam Nisan. Pseudorandom generators for space-bounded computation, 1992.