Theoretician's Toolkit **Spring 2019**

Lecture 9: Pseudorandomness

Scribe: Jonathan Liu 4/29/2019

9.1 Why do we care about randomness?

9.1.1 Randomness is useful.

Let's start by motivating why we should even care about randomness in our algorithms. Say you're given an algebraic expression for a (finite) polynomial p and want to determine whether or not it is identically zero.

To solve this deterministically, we would need to evaluate the expression completely, which can be take up to exponential time in the length of the expression if it is complicated enough. In fact, we don't yet have a subexponential deterministic algorithm to determine whether $p \equiv 0$.

In the face of this struggle, we turn to randomness and hope to leverage properties of polynomials to our advantage. Note that any polynomial has finite degree and thus a finite number of zeroes. If it happens that the polynomial is indeed not zero, then we know that $p(x) \neq 0$ for almost all x, which means that if we choose a random input r and evaluate the polynomial, we can be assured that if $p \not\equiv 0$ then $p(r) \neq 0$ with very high probability. This evaluation time is linear in the length of the expression, and we can even repeat the process a polynomial number of times to arbitrarily decrease the error probability.

9.1.2 Randomness is hard to generate.

Almost by definition, it is impossible to write an algorithm to generate truly random numbers. After all, random numbers shouldn't follow any set of rules or patterns, but algorithms are literally sets of rules for a computer to follow. "True" randomness may not even be achievable philosophically (if you believe the world to be completely deterministic), but the closest we can get right now is by looking for non-algorithm sources of entropy and drawing randomness from them. For example, Linux's /dev/random developed by Theodore Ts'o creates an "entropy pool" by drawing randomness from timings of computer hardware events (keystrokes, mouse movements, etc.) and using the entries in that pool as random seeds.

9.2 A class for randomized algorithms

As shown, there are algorithms that use randomness to significantly reduce the runtime of an algorithm while allowing only a small amount of error. We should discuss these in more depth, but before we do, it will be useful to formalize them.

As a baseline, we can define the complexity class P as follows:

Definition 9.1 (P)**.** *The class of decision problems that can be solved deterministically in polynomial time.*

Now, we can introduce randomness with the complexity class BPP (Bounded Probabilistic Polynomial).

Definition 9.2 (BPP)**.** *The class of decision problems that can be solved probabilistically in polynomial time, with error probability bounded by* $1/3$*.*

To reiterate, the class BPP allows the solution algorithm to make randomized decisions in constant time, and, for all inputs, the probability of outputting the correct answer is greater than $2/3$. Note that this constant can be replaced with any constant greater than $1/2$, because we can repeating the algorithm a linear number of times and taking the majority outcome from the repetitions causes the error probability to decrease exponentially as long as the probability of success is greater than the probability of failure (we can prove this with Chernoff bounds).

9.3 Does BPP = P?

Whether or not P = BPP is still an open problem! Clearly, P $\subset$ BPP. After all, BPP allows randomness but doesn't dictate that randomness must be used meaningfully in the algorithm implementation. The interesting question, then, is whether randomness significantly improves the strength of an algorithm.

One note is that if we could generate truly random numbers in polynomial time, the debate would be meaningless. Unfortunately, as described earlier, we don't have an algorithm to generate truly random numbers at all, let alone one that takes polynomial time.

9.3.1 Hardness vs. Randomness

Many theorists believe that the two classes are equal, in part because we don't have many examples of problems where we know of a randomized poly-time algorithm but not of a deterministic poly-time algorithm. We believed primality testing to be one of these problems until 2002, when Manindra Agrawal, Neeraj Kayal, and Nitin Saxena discovered a deterministic poly-time primality test [AKS02]. However, polynomial factorization and classification (as described above) are examples or such problems, as is prime-finding.

Perhaps one of the most convincing arguments for the equality is the following result by Impagliazzo and Wigderson:

Theorem 9.3 (IW97)**.** *If no* NP*-complete problem can be solved by a circuit of size* $2^{o(n)}$*, then* BPP = P*.*

While we'd like to think that randomness is a strong tool for algorithms, it's hard to argue with the claim that NP-complete problems are difficult and that we don't have any way to solve them quickly. This bring an interesting philosophical debate within computational complexity theory: are we more convinced that randomness is useful, or are we more convinced in the hardness of NP problems?

9.3.2 Derandomization

This discussion begs the next question: if we can't actually generate random numbers algorithmically, how close can we get? Do we even need truly random numbers to run BPP algorithms if randomness isn't that useful?

We make the following claim based on intuition, and will spend the rest of the section discussing it in more detail:

Claim 9.4. *If we can write a polynomial-time algorithm that's "close enough" to randomness, then* $\mathsf{BPP} = \mathsf{P}$.

To some extent, we should believe this to be true. A formal definition of "close enough" will be necessary before we can argue more rigorously about this, but on the basis that any poly-time algorithm can use at most a polynomial amount of randomness, we should think that if a poly-time randomness generator creates a distribution different from uniform by an exponentially small amount, then it should be good enough for any poly-time solver.

We note momentarily that if $\mathsf{BPP} = \mathsf{P}$, then randomness may not be as useful for algorithms as we currently think it is.

9.3.3 Pseudorandom Generators

Let's begin by introducing the object we'll be talking about.

Definition 9.5 (Pseudorandom Generator (PRG))**.** *A Pseudorandom generator is a function*

$$f : \{0,1\}^\ell \to \{0,1\})^n,$$

generally with $n > \ell$*. We can think of the function as taking in a seed of length* ℓ *and outputting a sequence of* n *pseudorandom bits.*

One immediate question to ask is how we can analyze the strength of a pseudorandom generator. To do so, we introduce ϵ-fooling.

Definition 9.6 (ϵ-fooling)**.** *A PRG* $G : \{0,1\}^\ell \to \{0,1\}^n$ *ϵ-fools a function* $f : \{0,1\}^n \to \{0,1\}$ *if*

$$\left| \Pr_{s \in \{0,1\}^\ell}[f(G(S)) = 1] - \Pr_{x \in \{0,1\}^n}[f(x)] \right| < \epsilon.$$

We can think of f as a function answering the question "Will the probabilistic poly-time algorithm return a valid answer if x is the randomness used?", so the *PRG* is said to ϵ-fool the function if it causes behavior at most ϵ different from true randomness.

Generally, we don't really care about looking at ϵ-fooling PRGs for single functions. Instead, we want to look at the behavior of a PRG over a class of functions, to see, for that particular problem, if the PRG we have selected mimics randomness well enough. Clearly, problems that rely more on randomness will have a harder time being mimicked by a PRG.

9.3.4 A step towards derandomization

Let's look at the power of ϵ-fooling PRGs, and what we can do with them. Assume the existence of a PRG G with seed length $\ell = O(\log n)$ that 0.4-fools every possible polynomial circuit. This is certainly a large value of ϵ, and for all we know, we wouldn't be able to say that this PRG mimics the uniform distribution well.

Let's see what we can do with this anyway. Let's pick any arbitrary BPP algorithm A, and assume that A is correct with probability 0.99 (remember that we can amplify this constant in polynomial time). Now, if we use G as its randomness generator, note by the previous definition that for any input x,

$$|\Pr_{s\in\{0,1\}^\ell}[A(x,G(s))=1] - \Pr_{r\in\{0,1\}^n}[A(x,r)=1] < 0.4.$$

Then,

$$\begin{aligned}\Pr_{s\in\{0,1\}^\ell}[A(x,G(s)) \text{ is wrong}] &\leq \Pr[G(s) \text{ causes the wrong output}] + \Pr[\text{uniform randomness gives wrong output}]\\ &\leq 0.4 + 0.01\\ &\leq 0.41.\end{aligned}$$

This leads us to two conclusions. First, by sacrificing some of the error prevention, we have greatly decreased the amount of randomness we need; we went from n bits of randomness to $\log(n)$ bits of randomness! Secondly, and more importantly, we know that at most 41% of the initial seeds produce incorrect results. However, there are only $2^{O(\log n)} = poly(n)$ total possible seeds. We can simply try them all in polynomial time and return the majority outcome, and this output will never be wrong! This is a massive (albeit still polynomial) blowup in runtime, but it reduces a polynomial probabilistic algorithm to a polynomial deterministic algorithm, thereby derandomizing it.

An important note is that, in doing so, this PRG would have successfully collapsed BPP into P. We have finally found a formalized version of Claim 9.4, restated below with our new findings.

Claim 9.7. *If there exists a PRG $G : \{0,1\}^{O(\log n)} \to \{0,1\}^n$ that 0.4-fools every probabilistic poly-time function, then* BPP = P.

9.4 Pseudorandomness in Objects

Another interesting focus of study related to pseudorandomness is the pseudorandom generation of objects like graphs or sequences where most objects would satisfy certain desirable properties, If it's the case that most objects satisfy these properties, then certainly most randomly generated ones do too. This is closely tied to the Probabilistic Method from Week 2; the implicit generation of objects with certain properties derived from probability rather than from construction. The explicit generation of objects with these "random" properties will be the focus of this section.

9.4.1 Expanders

We've talked about expander graphs previously in our discussion of Spectral graph theory as well as Markov chains. These graphs are undirected graphs that are both sparse and well-connected. For simplicity, we'll assume that the graph $G = (E, V)$ is a d-regular graph.

Let's formalize these properties. We would like these graphs to have relatively few edges, but for sets to have many neighbors. Let's call $\Gamma(S)$ the set of unique neighbors of a set S.

Definition 9.8 (Well-connectedness)**.** *A d-regular graph on n vertices is well-connected if, for every subset S of at most $n/2$ vertices, $|\Gamma(S)| \geq (5/4)|S|$.*

This follows our intuition that the graph is "well-connected" in the sense that every vertex is "close" to many other points. Note that the constant $5/4$ is chosen arbitrarily.

Now, we can get right into the power of probabilistically constructing these graphs.

Claim 9.9. *There exists a d for any n such that one can generate a d-regular graph randomly in a way such that it is an expander with high probability.*

Proof. We'll construct the expander graph by simply randomly constructing d perfect matchings and overlaying them onto a graph.

Now, let's look at the probability that the graph we have constructed is not well-connected. For this to be the case, there must be some subset S of k vertices and subset T of $5k/4$ vertices such that $\Gamma(S) \subset T$. For any particular matching, any edge originating from S has probability $\frac{5k/4}{n}$ of being in T, and the probability actually decreases if other edges from S are in T, so we have

$$\Pr[\Gamma_{\text{matching}}(S) \subset T] \leq \left(\frac{5k/4}{n}\right)^{k/2}.$$

Therefore, as there are d perfect matchings to consider,

$$\Pr[\Gamma(S) \subset T] \leq \left(\frac{5k/4}{n}\right)^{dk/2}.$$

Now, there are $\binom{n}{k}$ candidates for S and $\binom{n}{5k/4}$ candidates for T, so by union bound we have that the probability that some subset of size k does not have enough neighbors is

$$\begin{aligned}
\binom{n}{k}\binom{n}{5k/4}\left(\frac{5k/4}{n}\right)^{dk/2} &\leq \left(\frac{ne}{k}\right)^{k}\left(\frac{ne}{5k/4}\right)^{5k/4}\left(\frac{5k/4}{n}\right)^{dk/2} \\
&\leq \left(\frac{en}{k}\right)^{9k/4}\left(\frac{5k}{4n}\right)^{dk/2} \\
&\leq \left(\frac{en}{k}\right)^{9k/4}\left(\frac{5k}{4n}\right)^{9k/4}\left(\frac{5k}{4n}\right)^{(dk/2)-(9k/4)} \\
&\leq \left(\frac{5e}{4}\right)^{9k/4}\left(\frac{5k}{4n}\right)^{(dk/2)-(9k/4)} \\
&\leq \left(\frac{5e}{4}\right)^{9k/4}\left(\frac{5}{8}\right)^{\frac{k}{4}(2d-9)} \\
&\leq \left((2e)^{9}\left(\frac{5}{8}\right)^{2d}\right)^{k/4}
\end{aligned}$$

For sufficiently large d (the inside is less than one for $d > 16$) this clearly approaches zero. We sum over all values of $1 \leq k \leq n/2$, and find that this value is still extremely low for any n, and that actually increasing n only marginally increases the chance of failure. Therefore, more graphs generated this way will be well-connected. □

We have thus shown that not only are expander graphs common, but in fact *most* random graphs are expander graphs. If this is the case, one would think that deterministically generating an expander graph with these properties for any n should be easy. Surprisingly enough, we have no explicit constructions that fulfill desired properties as well as randomly generated graphs do! This problem has been described by Howard Karloff as "finding hay in a haystack;" we know that the graphs are everywhere we look, and yet the ability to generate one with those properties still eludes us.

Below are some other interesting cases of objects modeling the same principle.

9.4.2 Ramsey Graphs

The Ramsey Graph is another graph mentioned during our exploration of the probabilistic method. A graph is r-Ramsey if it contains no clique of size r and no independent set of size r. In 1947, Erdös proved that almost every graph on n vertices is $3 \log n$-Ramsey, and yet to this day we have no polynomial time algorithm for generating a random graph that is $3 \log n$-Ramsey.

9.4.3 Universal Traversal Sequences

Let's assume we have some graph $G = (E, V)$ where G is d-regular, where the edges are labelled $1, ..., d$ in a way that each vertex is adjacent to exactly one edge of each label. We'd like to find a sequence of moves such that, for any graph, any labelling of its edges, and any starting vertex, the sequence of moves can be guaranteed to reach every possible vertex in the component.

First, we will call back a lemma from our lecture on Electrical Flows:

Lemma 9.10. *The cover-time of d-regular graph with n vertices is $dn(n-1)$.*

This, as a result, is already astounding. It tells us that on expectation, a random walk only needs to take $dn(n-1)$ steps to reach every vertex on the graph. Thus, if we want a path on a graph that hits every single vertex, we only need to sample a few $dn(n-1)$-step paths, and we have a very high probability of finding one.

Here's another frustrating result about Universal Traversal Sequences:

Theorem 9.11. *There exist universal traversal sequences of length $O(n^3 \log n)$.*

Proof. We define our sequence as a sequence of "steps," where each step is simply a choice of enumerated edge. Now, we *randomly* select a sequence S of length

$$L = 2dn(n-1)(dn+2)(\log n).$$

We then split L into $(dn+2)(\log n)$ segments of length $2dn(n-1)$.

Denote by $X_{G,v}$ the indicator variable for whether or not all vertices of d-regular graph G with n vertices are covered by our sequence when starting at vertex v, and denote $Y := \sum X_{G,v}$ for all possible graphs and starting vertices.

By our lemma, for any segment of S that we select, the probability that this segment does not traverse every

vertex in G is bounded by Markov's inequality:

$$\Pr[\text{a cover of G} \geq 2dn(n-1)] \leq \frac{E[\text{cover}}{2dn(n-1)} = \frac{1}{2},$$

so the probability that this segment does not traverse every vertex in G is less than $1/2$. Therefore,

$$E[X_{G,v}] \leq 2^{-(dn+2)(\log n)} = n^{-(dn+2)}.$$

Note then that there must be less than n^{dn} possible d-regular graphs because each vertex can be adjacent to less than n^d possible combinations of vertices, and each graph has n possible starting vertices, so

$$E[Y] = \sum E[X_{G,v}] \leq n^{dn} n n^{-(dn+2)} = n^{-1} < 1,$$

so there must be some sequence for which $Y = 0$. This sequence has length $O(2dn(n-1)(dn+2)(\log n)) = O(n^3 \log n)$, as d is a constant. □

What is perhaps most frustrating is that we currently have no poly-time algorithm for producing a universal traversal sequence of any polynomial length, let alone the relatively short bound produced by simply choosing a path at random.