Spectral Graph Theory **Fall 2019**

Lecture 7: Pseudorandom Generators via Random Walks

Scribes: Jonathan Liu 11/7/2019

Based on Daniel Spielman's scribe notes.

7.1 Pseudorandom Generators

Let's begin by introducing, in abstract, the object we'll be talking about today.

Definition 7.1 (Pseudorandom Generator (PRG))**.** *A Pseudorandom generator is a function*

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

generally (but not always) with $\ell < n$. We can think of the function as taking in a seed of length ℓ and outputting a sequence of n bits that we hope are "random enough" to be mostly acceptable as truly random bits, despite having a deterministic generation based on a shorter input seed.

Before we continue, we should talk about why this even matters.

7.1.1 Why do we care about (pseudo)randomness?

7.1.1.1 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.

7.1.1.2 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. Cryptographic primitives (one-way functions in particular) have been used to construct PRGs as well, but those primitives have not been shown to exist and our candidates for them may still be breakable.

7.1.1.3 Repeatability is sometimes useful.

If you've developed a randomized algorithm and you wrote code to simulate it but you need to regenerate the same conditions in order to debug it, it's helpful to have the ability to easily repeat the algorithm by using the same small set of input bits for the PRG.

7.2 Expander Graphs

We talked about expander graphs last week, but just to give a brief recap, expander graphs are graphs with generally high conductance and sparsity. We'll focus on d-regular graphs, and we can characterize expander graphs as graphs with some bounded second adjacency eigenvalue. For today, we'll require that our graph's adjacency matrix meets the inequality

$$|\mu_i| < \frac{1}{10}d, \quad \forall i \geq 2.$$

These graphs are very useful due to their high conductance despite low degree. We do know of explicit constructions of expander graphs for any degree d, but it turns out that randomly selected d-regular graphs are already expanders with high probability, and the explicit generation of expander graphs generally doesn't even meet the expectations set by randomly selected graphs.

Today, we'll be using the fact that expander graphs mimic complete graphs very well to create a pseudorandom generator.

7.3 An Overview of our Problem

It's very clear that the ability to repeatedly randomized algorithms on new randomness is valuable, but using a new set of random bits for each runthrough can be large and every random bit used is expensive. For this reason, we'd like to be able to repeat an algorithm "pseudo-randomly" - enough times, and with enough unpredictability between new inputs, that we simulate truly randomizing the input bits without needing to spend many bits each runthough.

7.3.1 The Scenario

We have a randomized algorithm $\mathcal{A}$ that runs on r random bits, and returns a yes/no answer to some question. For any input to the question, we can guarantee that the algorithm outputs the correct result for 99% of the possibilities of the r random bits. Note that this is *very* different from outputting the correct result on 99% of inputs; if our algorithm works for 99% of inputs but fails on the other 1%, is it even a proper algorithm?

While 99% is a pretty strong bound, we'd like to do better by repeating the experiment over and over and taking the final answer to be the majority of the outputs. However, we assume r is reasonably large, so we don't want to spend too many random bits running new tries of the algorithm.

7.3.2 Our Tools

As mentioned earlier, we'll be using an expander graph where

$$|\mu_i| < \frac{1}{10}d, \quad \forall i \geq 2.$$

In particular, we'll take one such graph with degree $d = 400$. We'll see how we can construct these graphs next week; in the meantime, you can just take our word for it.

Now, the vertices of our graph will be each of the 2^r possibilities for words created by the values of the r bits of randomness our algorithm uses. This is a natural set of vertices to create an expander for: if we have a complete graph on these vertices, any random walk on the graph would clearly give us a randomly generated series of words in 2^r, so an expander graph on these vertices will hopefully perform the same job.

One important distinction we must make is that this graph takes exponential space to keep track of, so for our plan to have any feasibility the graph needs to be succinctly representable. In particular, if given any vertex v and edge number $n < 400$, it should be quick and easy for us to find the vertex connected to v by its nth edge. You'll just have to trust that we know how to do this.

7.3.3 Our Goal

The first runthrough of our algorithm will have r randomly generated bits; given that we don't actually have any specifications on what the algorithm is doing, it's best to be safe and start at a completely random point. Let's define X to be the set of random strings in 2^r that cause an incorrect output. We want to show that, using our expander graph with $d = 400$, taking a random walk of length t on the graph 2^r and evaluating the function at each vertex ,

$$\Pr[\text{Majority of outcomes is incorrect}] \leq \left(\frac{2}{\sqrt{5}}\right)^{t+1}.$$

The important thing to note here is that, for $d = 400$, generating a random step to take requires only 9 bits of randomness! Thus, instead, of r bits of randomness per run, we can use only 9, and still find an exponentially decreasing bound on the chance that our algorithm fails us.

Before we begin, we'll need to briefly introduce one more tool.

7.3.4 Matrix Norms

We'd like a measure of how much a matrix can affect the magnitude of a vector. This is pretty analogous to the Rayleigh Quotient, but in a more general sense, because we don't require the matrix to be symmetric (or even be square for that matter). We define the Matrix Norm for this purpose as follows:

$$||M|| = \max_v \frac{||M\vec{v}||}{||\vec{v}||}.$$

This is a fairly natural definition, and we can actually see that for symmetric N this value is just the largest eigenvalue.

One idea we'll need is that matrices are submultiplicative:

Claim 7.2. *For any matrices M_1, M_2, we have*

$$||M_1 M_2|| \leq ||M_1|| \cdot ||M_2||.$$

Proof. Note that

$$\begin{aligned} ||M_1 M_2|| &= \max_v \frac{||M_1 M_2 \vec{v}||}{||\vec{v}||} \\ &= \max_v \frac{||M_1(M_2\vec{v})||}{||M_2\vec{v}||} \cdot \frac{||M_2\vec{v}}{||\vec{v}||} \\ &\leq \max_{w \in \text{range}(M_1)} \frac{||M_1\vec{w}}{||\vec{w}||} \cdot ||M_2|| \\ &\leq ||M_1|| \cdot ||M_2||. \end{aligned}$$

□

7.4 The Work

7.4.1 Notation and Terminology

We'll use the following notation in our proof:

- As mentioned earlier, X is the set of strings in 2^r that produce a bad result. Y is the rest of the strings, the ones producing a good result. Remember that $|X| \leq \frac{2^r}{100}$.
- We denote by $\vec{I}_X, \vec{I}_Y$ the indicator vectors for X and Y, and denote by D_X, D_Y the diagonal matrices corresponding to these vectors.
- We denote p_i by the probability distribution of the location of the walk after i steps, and $p_0 = \frac{1}{n}\vec{1}$ as the initial probability distribution, as we start with a uniform choice of vector.
- $W = \frac{1}{d}A$ is the transition matrix of the random walk. Its eigenvalues are $\omega_1 \ldots \omega_n$, with $|\omega_i| \leq 1/10$ for $i \geq 2$.

7.4.2 Proof Outline

Note that for any probability distribution $\vec{p}$ on the set of points, $D_X\vec{p}$ is the probability vector of only points in X, and $\vec{1} \cdot D_X\vec{p}$ is the probability that this distribution is in X. Note then that the probability that a random walk starts in a point in X then travels to a point in Y is

$$\vec{1} \cdot D_Y W D_X \vec{p}_0.$$

With this in mind, we'll look at paths through the graph by classifying each of the points on the path, and use matrix bounds on the probability of a path of that form occurring. With a bound on the probability of a "bad" path, we'll union bound over all "bad" paths.

7.4.3 Bounding Steps into X

We'll start by bounding the norm $||D_X W||$, which allows us to bound the probability that a step on the walk goes into X.

Claim 7.3. $||D_X W|| \leq \frac{1}{5}$.

Proof. Remember that we're looking for an upper bound on

$$\frac{|D_X W\vec{v}|}{||\vec{v}||}$$

for any vector $\vec{v}$.

First, because W is symmetric, it has a set of orthonormal eigenvalues/eigenvectors, with largest eigenvalue 1 (with eigenvector $\vec{1}$) and the rest of the eigenvalues are all less than 1/10 by our criteria for expanders. (Remember, after all, that $W = \frac{1}{d}A$.) We'll take any vector $\vec{v}$ and write it as its sum of eigenvectors:

$$\begin{aligned}\vec{v} &= \sum_{i=1} c_i\phi_i \\ &= c_1\vec{1} + \sum_{i=2} c_i\phi_i.\end{aligned}$$

With this rewriting, we can see that

$$D_X W\vec{v} = c_1 D_X W\vec{1} + D_X W \sum_{i=2} c_i\phi_i.$$

Now, we'll bound each of these values individually.

First, note that $W\vec{1} = \vec{1}$, so $D_X Q\vec{1} = I_X$, so

$$||c_1 D_X W\vec{1}|| = ||c_1 I_X|| = c_1\sqrt{|X|} \leq c_1\frac{\sqrt{n}}{10} = ||c_1\vec{1}||/10 \leq ||\vec{v}||/10.$$

Next, let's look at $\sum_{i=2} c_i\phi_i$. Note by our criteria for an expander that $\omega_i \leq 1/10$ so

$$||W\left(\sum_{i=2} c_i\phi_i\right)|| = ||\sum_{i=2} c_i\omega_i\phi_i|| \leq \frac{1}{10}\sqrt{\sum_{i\geq 2}(c_i)^2} = \frac{1}{10}\sqrt{\sum_{i\geq 2}||c_i\phi_i||^2} = \frac{1}{10}||\sum_{i\geq 2} c_i\phi_i|| \leq \frac{1}{10}||\vec{v}||.$$

Now, note because Matrix norms are submultiplicative that $||D_X W\vec{y}|| \leq ||D_X|| \cdot ||W\vec{y}|| \leq ||W\vec{y}||$, as $||D_X|| \leq 1$.

Combining these terms, we have

$$||c_1 D_X W\vec{1}|| + ||c_1 D_X W\sum_{i=2} c_i\phi_i|| \leq \frac{1}{10}||\vec{x}|| + \frac{1}{10}||\vec{x}|| = \frac{1}{5}||\vec{x}||,$$

as desired. □

Now that we have a bound on the effect of stepping into X on the probability of the entire walk, we can work on bounding the probability of an entire walk of t steps.

Claim 7.4. *For a set $R \subseteq [t]$, the probability that the walk is in X at precisely the times in R is at most $(1/5)^|R|$.*

Proof. Note as mentioned earlier that a path can be represented as

$$\vec{1} \cdot \prod_{i\geq 1}^{t}(D_i W)D_0\vec{p}_0,$$

where $D_i = D_X$ if step i is in a bad vertex and $D_i = D_Y$ is in a good vertex. Note that because our graph is d-regular, $\vec{p}_0 = W\vec{p}_0$, so this path can be written as

$$\vec{1} \cdot \prod_{i\geq 0}^{t}(D_i W)\vec{p}_0.$$

Now, because this product represents the probability of a path with the given layout R, we'd like to find an upper bound for it. Note that

$$\begin{aligned}
\vec{1} \cdot \prod_{i\geq 0}^{t}(D_i W)\vec{p}_0 &\leq ||\vec{1}|| \cdot ||\prod_{i\geq 0}^{t}(D_i W)|| \cdot ||\vec{p}_0|| \\
&\leq ||\vec{1}|| \cdot ||\vec{p}_0|| \cdot \prod_{i\geq 0}^{t}||(D_i W)|| \\
&\leq \sqrt{n} \cdot \frac{1}{\sqrt{n}} \cdot ||D_X W||^{|R|} \cdot ||D_Y W||^{t-|R|} \\
&\leq ||D_X W||^{|R|} \cdot ||D_Y W||^{t-|R|} \\
&\leq ||D_X W||^{|R|} \\
&\leq (1/5)^{|R|}.
\end{aligned}$$

□

To finish the proof, we'll just take a simple union bound. There are 2^{t+1} possible sequences of X and Y, and for each one of them with more than $(t+1)/2$ points in X, we can see that

$$\begin{aligned}\Pr[\text{majority is in } X] &\leq 2^{t+1} \cdot \frac{1}{5}^{(t+1)/2} \\ &= \left(\frac{2}{\sqrt{5}}\right)^{t+1}.\end{aligned}$$