**Spectral Graph Theory** Lecture 9

# PSRGs via Random Walks on Graphs

*Daniel A. Spielman* September 30, 2009

## 9.1 Overview

There has been a lot of work on the design of Pseudo-Random Number Generators (PSRGs) that can be proved to work for particular applications. In this class, we will see one of the foundational results in this area, namely Impagliazzo and Zuckerman's [IZ89] use of random walks on expanders to run the same algorithm many times. We are going to preform a very crude analysis that is easy to present in class. Rest assured that much tighter analyses are possible and much better PSRGs have been constructed since.

## 9.2 Why Study PSRGs?

Pseudo-random number generators take a seed which is presumably random (or which has a lot of randomness in it), and then generate a long string of random bits that are supposed to act random. We should first discuss why we would actually want such a thing. I can think of two reasons.

1. Random bits are scarce. This might be surprising. After all, if you look at the last few bits of the time that I last hit a key, it is pretty random. Similarly, the low-order bits of the temperature of the processor in my computer seem pretty random. While these bits are pretty random, there are not too many of them.

   Many randomized algorithms need *a lot* of random bits. Sources such as these just do not produce random bits with a frequency sufficient for many applications.

2. If you want to re-run an algorithm, say to de-bug it, it is very convenient to be able to use the same set of random bits by re-running the PSRG with the same seed. If you use truly random bits, you can't do this.

You may also wonder how good the standard pseudo-random number generators are. The first answer is that the default ones, such as `rand` in C, are usually terrible. There are many applications, such as those in my thesis, for which these generators produce behavior that is very different from what one would expect from truly random bits (yes, this is personal). On the other hand, one can use cryptographic functions to create bits that will act random for most purposes, unless one can break the underlying cryptography [HILL99]. But, the resulting generators are usually much slower than the fastest pseudo-random generators. Fundamentally, it comes down to a time-versus-quality tradeoff. The longer you are willing to wait, the better the pseudo-random bits you can get.

## 9.3 Today's Application : repeating an experiment

Today, we are going to consider a very simple situation in which one wants to run the same randomized algorithm many times. Imagine that you have a randomized algorithm $A$, that returns "yes"/"no" answers, and which is usually correct. In particular, imagine that 99% of the time it returns the correct answer. So, whether or not it returns the correct answer is a function of the random bits it is fed, rather than the input. Sometimes 99% accuracy is enough, but sometimes it is not. We can boost the accuracy by running the algorithm a few times, and taking the majority vote of the answers we get back[1].

Since we will not make any assumptions about the algorithm, we will use truly random bits the first time we run it. But, we will show that we only need 9 new random bits for each successive run. In particular, we will show that if we use our PSRG to generate bits for $t+1$ runs, then the probability that majority answer is wrong decreases exponentially in $t$.

## 9.4 The Random Walk Generator

Let $r$ be the number of bits that our algorithm needs for each run. So, the space of random bits is $\{0,1\}^r$. Let $X \subset \{0,1\}^r$ be the settings of the random bits on which the algorithm gives the wrong answer, and let $Y$ be the settings on which it gives the correct answer.

Our pseudo-random generator will use a random walk on an expander graph whose vertex set is $\{0,1\}^r$. The expander graph will be $d$-regular, for some constant $d$. Let $A$ be the adjacency matrix of the graph, and let its eigenvalues be $d = \alpha_1 > \alpha_2 \geq \cdots \geq \alpha_n$. One property that we will require of the expander graph is that

$$\frac{|\alpha_i|}{d} \leq 1/10, \tag{9.1}$$

for all $i$. We will see how to construct such graphs next week. In the meantime, let me promise you that such graphs exist with $d = 400$. Degree 400 seems sort of big, but we will see that it is reasonable for our application.

For the first run of our algorithm, we will require $r$ truly random bits. We treat these bits as a vertex of our graph. For each successive run, we choose a random neighbor of the present vertex, and feed the corresponding bits to our algorithm. That is, we choose a random $i$ between 1 and 400, and move to the $i$th neighbor of the present vertex. Note that we only need $\log_2 400 \approx 9$ random bits to choose the next vertex. So, we will only need 9 new bits for each run of the algorithm.

One thing to be careful about here is that we will need a very concise description of the graph. As the graph has $2^r$ vertices, and $r$ may be large, we certainly wouldn't want to store the whole graph, let alone the adjacency matrix. Instead, we need a fast procedure that given the name of a vertex $v \in \{0,1\}^r$ and an $i \leq d$ will quickly produce the name of the $i$th neighbor of $v$. To see how such a thing might be possible, consider the hypercube graph on $\{0,1\}^r$. It has degree $d = r$, and given a $v \in \{0,1\}^r$ and an $i \leq d$, we can obtain the name of the $i$th neighbor of $v$ by just flipping the $i$th bit of $v$. For now, I promise you that we know expanders with similarly concise descriptions.

[1] Check for yourself that running it twice doesn't help

## 9.5 Formalizing the problem

Assume that we are going to run our algorithm $t+1$ times. So, our pseudo-random generator will begin at a truly random vertex $v$, and then take $t$ random steps. Recall that we defined $X$ to be the set of vertices on which the algorithm gets the answer wrong, and we assume that $|X| \leq 2^r/100$. If we report the majority of the $t+1$ runs of the algorithm, we will return the correct answer as long as the random walk is inside $X$ less than half the time. To analyze this, let $v_0$ be the initial random vertex, and let $v_1, \ldots, v_t$ be the vertices produced by the $t$ steps of the random walk. Let $T = \{0, \ldots, t\}$ be the time steps, and let $S = \{i : v_i \in X\}$. We will prove

$$\Pr\left[|S| > t/2\right] \leq \left(\frac{2}{\sqrt{5}}\right)^{t+1}.$$

To begin our analysis, recall that the initial distribution of our random walk is $\boldsymbol{p}_0 = \mathbf{1}/n$. Let $\chi_X$ and $\chi_Y$ be the characteristic vectors of $X$ and $Y$, respectively, and let $D_X = \mathtt{diag}(X)$ and $D_Y = \mathtt{diag}(Y)$. Let

$$W = \frac{1}{d}A \tag{9.2}$$

be the transition matrix of the ordinary random walk on $G$. Last lecture I said that I would always use $W$ to denote the transition matrix of the lazy random walk. But, this is not the lazy random walk. So, I lied. Let $\omega_1, \ldots, \omega_n$ be the eigenvalues of $W$. By (9.2) and assumption (9.1), $|\omega_i| \leq 1/10$.

Let's see how we can use these matrices to understand the probabilities under consideration. For a probability vector $\boldsymbol{p}$ on vertices, the probability that a vertex chosen according to $\boldsymbol{p}$ is in $X$ may be expressed

$$\chi_X^T \boldsymbol{p} = \mathbf{1}^T D_X \boldsymbol{p}.$$

The second form will be more useful, as

$$D_X \boldsymbol{p}$$

is the vector obtained by zeroing out the events in which the vertices are not in $X$. If we then want to take a step in the graph $G$, we multiply by $W$. That is, the probability that the walk starts at vertex in $X$, and then goes to a vertex $i$ is $\boldsymbol{q}(i)$ where

$$\boldsymbol{q} = W D_X \boldsymbol{p}_0.$$

Continuing this way, we see that the probability that the walk is in $X$ at precisely the times $i \in S$ is

$$\mathbf{1}^T D_{Z_t} W D_{Z_{t-1}} W \cdots D_{Z_2} W D_{Z_1} \boldsymbol{p}_0,$$

where

$$Z_i = \begin{cases} X & \text{if } i \in S \\ Y & \text{otherwise.} \end{cases}$$

We will prove that this probability is at most $(1/5)^{|S|}$. It will then follow that

$$\begin{aligned}\Pr\left[|S| > t/2\right] &\leq \sum_{|S|>t/2} \Pr\left[\text{the walk is in } X \text{ at precisely the times in } S\right] \\ &\leq 2^{t+1}\left(\frac{1}{5}\right)^{(t+1)/2} \\ &= \left(\frac{2}{\sqrt{5}}\right)^{t+1}.\end{aligned}$$

## 9.6 Matrix Norms

Recall that the operator norm of a matrix $M$ (also called the 2-norm) is defined by

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

When $M$ is symmetric, the 2-norm is just the largest absolute value of an eigenvalue of $M$ (prove this for yourself). It is also immediate that

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

You should also verify this yourself. As $D_X$, $D_Y$ and $W$ are symmetric, it is easy to show that each has norm at most 1.

**Warning 9.6.1.** *While the largest eigenvalue of a walk matrix is* 1*, the norm of a walk matrix can be larger than* 1*. For instance, consider the walk matrix of the path on* 3 *vertices. Verify that it has norm* $\sqrt{2}$*.*

Our analysis rests upon the following bound on the norm of $D_X W$.

**Lemma 9.6.2.**

$$\|D_X W\| \leq 1/5.$$

Let's see why this implies the theorem. For any set $S$, let $Z_i$ be as defined above. As $\boldsymbol{p}_0 = W\boldsymbol{p}_0$, we have

$$\mathbf{1}^T D_{Z_t} W D_{Z_{t-1}} W \cdots D_{Z_2} W D_{Z_1} \boldsymbol{p}_0 = \mathbf{1}^T \left(D_{Z_t} W\right)\left(D_{Z_{t-1}} W\right)\cdots\left(D_{Z_1} W\right)\boldsymbol{p}_0.$$

Now,

$$\left\|D_{Z_{t-1}} W\right\| \leq \begin{cases} 1/5 & \text{for } i \in S, \text{ and} \\ 1 & \text{for } i \notin S. \end{cases}$$

So,

$$\left\|\left(D_{Z_t} W\right)\left(D_{Z_{t-1}} W\right)\cdots\left(D_{Z_1} W\right)\right\| \leq (1/5)^{|S|}.$$

As $\|p_0\| = 1/\sqrt{n}$ and $\|\mathbf{1}\| = \sqrt{n}$, we may conclude

$$\begin{aligned}\mathbf{1}^T\left(D_{Z_t} W\right)\left(D_{Z_{t-1}} W\right)\cdots\left(D_{Z_1} W\right)\boldsymbol{p}_0 &\leq \left\|\mathbf{1}^T\right\| \left\|\left(D_{Z_t} W\right)\left(D_{Z_{t-1}} W\right)\cdots\left(D_{Z_1} W\right)\boldsymbol{p}_0\right\| \\ &\leq \left\|\mathbf{1}^T\right\| (1/5)^{|S|} \|\boldsymbol{p}_0\| \\ &= (1/5)^{|S|}.\end{aligned}$$

## 9.7 The norm of $D_X W$

*Proof of Lemma 9.6.2.* Let $\boldsymbol{x}$ be any non-zero vector, and write

$$\boldsymbol{x} = c_1 \mathbf{1} + \boldsymbol{y},$$

where $\mathbf{1}^T \boldsymbol{y} = 0$. So,

$$\|\boldsymbol{x}\| = \sqrt{\left(c_1\sqrt{n}\right)^2 + \|\boldsymbol{y}\|^2} \geq \min\left(c_1\sqrt{n}, \|\boldsymbol{y}\|\right).$$

We also have

$$D_X W \boldsymbol{x} = c_1 D_X W \mathbf{1} + D_X W \boldsymbol{y}.$$

We know that $W\mathbf{1} = \mathbf{1}$, so

$$D_X W \mathbf{1} = \chi_X,$$

and

$$\|c_1 D_X W \mathbf{1}\| = c_1 \|\chi_X\| = c_1\sqrt{|X|} \leq c_1\sqrt{n}/10 \leq \|\boldsymbol{x}\|/10.$$

To bound the norm of the other term, recall that $W$ is symmetric, and that its leading eigenvector $\boldsymbol{v}_1$ is a constant vector. So, we may expand $\boldsymbol{y}$ in the other eigenvectors $\boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ as

$$\boldsymbol{y} = \sum_{i \geq 2} c_i \boldsymbol{v}_i, \qquad \text{where } c_i = \boldsymbol{v}_i^T \boldsymbol{y}.$$

We find

$$\|W\boldsymbol{y}\| = \left\| \sum_{i \geq 2} c_i \omega_i \boldsymbol{v}_i \right\| = \sqrt{\sum_{i \geq 2} (c_i \omega_i)^2} \leq (1/10)\sqrt{\sum_{i \geq 2} (c_i)^2} = (1/10)\|\boldsymbol{y}\|.$$

So,

$$\|D_X W \boldsymbol{y}\| \leq \|D_X\| \, \|W\boldsymbol{y}\| \leq (1/10)\|\boldsymbol{y}\| \leq (1/10)\|\boldsymbol{x}\|.$$

Summing these two terms, we find

$$\|D_X W \boldsymbol{x}\| \leq \|D_X W c_1 \mathbf{1}\| + \|D_X W \boldsymbol{y}\| \leq (1/10)\|\boldsymbol{x}\| + (1/10)\|\boldsymbol{x}\| = (1/5)\|\boldsymbol{x}\|.$$

□

## References

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[IZ89] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In IEEE, editor, *30th annual Symposium on Foundations of Computer Science, October 30–November 1, 1989, Research Triangle Park, North Carolina*, pages 248–253, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1989. IEEE Computer Society Press.