1 Last time

Last time we talked about random walks on graphs, and in particular random walks on undirected graphs. We saw that performing a random walk on an undirected graph would solve the undirected $s-t$ connectivity problem in randomized logspace. We also saw that the time it takes for a random walk on a graph (that’s undirected, $d$-regular and ergodic) to approach the stationary distribution (which is the uniform distribution over the nodes) within $\epsilon$ in $\ell_2$ norm takes time about $\log\frac{1}{\epsilon}/\log \lambda_2$.

2 Today

Today we will see how random walks can be used to save randomness in randomized algorithms while attaining very low error. Suppose we had a polynomial time randomized algorithm $A$ for a language $L \in \text{RP}$ with the following property:

- $x \in L$: $\Pr_w[A(x, w) \text{ doesn’t accept}] \leq 1/100$, and
- $x \notin L$: $\Pr_w[A(x, w) \text{ accepts}] = 0$, and
- $A$ uses $r(n)$ bits of randomness.

Previously, we saw two ways in which the error of this algorithm could be reduced to $2^{-k}$:

1. Run $A$ $k$ times, using new randomness for each repetition. This uses $r \cdot k$ random bits, and has running time $k \cdot T(n)$ (where $T(n)$ is the running time of $A$); or
2. Run $A$ on pairwise independent randomness. This uses $O(r + k)$ random bits, and has running time $O(2^k \cdot T(n))$.

We will describe a new method that combines the time efficiency of Method 1 with the randomness efficiency of Method 2, and this will be done via random walks on a special type of graph, called an expander graph.

3 A Special Kind of Graph

We will postulate that we can construct a family of undirected graphs $G_n = (V_n, E_n)$ with the following properties:

1. $G_n$ has constant degree (i.e. there is a constant $d$ such that the maximum degree of $G_n$ is bounded by $d$);
2. $G_n$ is regular;
3. $\lambda_2 \leq 1/10$ (where $\lambda_2$ is the second largest eigenvalue of the transition matrix corresponding to $G_n$);
4. $G_n$ is ergodic (this is redundant, because of the $\lambda_2$ property above);
5. $|V_n| = 2^r(n)$;

6. Given a node $v \in V_n$, the neighbors of $v$ can be computed efficiently (in $n$).

It turns out that it is possible to construct a graph family with all these magical properties: such constructions are called expander graph constructions. Here, we will take the construction of these graphs for granted. Importantly, any expander graph construction that has the above properties will suffice for the error reduction procedure we describe next.

4 Error reduction for $\mathcal{A}$

We describe the algorithm to reduce the error of $\mathcal{A}$:

- **Algorithm $\mathcal{A}'(x)$:**
  - Pick a random node $w$ in $G_n$.
  - Repeat $k$ times:
    * Run $\mathcal{A}(x, w)$. If $\mathcal{A}$ accepts, then halt and accept.
    * Otherwise, pick a random neighbor of $w$ and set $w$ equal to that neighbor.
  - If the algorithm has not accepted by this time, reject.

Clearly, the running time of $\mathcal{A}'$ is $O(k(T(n) + Q(n)))$, where $T(n)$ is the running time of $\mathcal{A}$ and $Q(n)$ is the time it takes to pick a random neighbor in $G_n$. We assume here that $Q(n)$ is negligible compared to $T(n)$.

The randomness efficiency is $r(n) + O(k)$: $r(n)$ bits of randomness are required to pick the initial start node $w$. Then, for each iteration of the loop, a random neighbor has to be picked. Since each node has $d$ neighbors and $d$ is constant, each step of the random walk requires $\log d = O(1)$ random bits.

We now argue that the error of $\mathcal{A}'$ is very low:

**Claim 1** The error probability of $\mathcal{A}'$ is at most $\frac{1}{5^r}$.

Before diving into the proof, let us spend a few moments to reflect on this algorithm. We have essentially applied a graph structure to the space of all possible random strings that could be used by the original algorithm $\mathcal{A}$. For a fixed input $x$, the set of bad random strings $B_x$ (random strings that cause $\mathcal{A}$ to err on $x$) contains at most $1/100$ of all the possible nodes. Because of the one-sided error property of $\mathcal{A}$, we only need to encounter just one good random string in order to succeed.

Our algorithm $\mathcal{A}'$ will take a random walk on this graph of random strings, and because of the $\lambda_2$ condition stipulated above, this graph has very good mixing properties: it is very unlikely for a random walk to be trapped within $B_x$. The following proof formalizes this intuition.

**Proof** Suppose input $x \notin L$. Then because of the soundness property of $\mathcal{A}$, $\mathcal{A}$ will never accept $x$, no matter the randomness. Thus $\mathcal{A}'$ will never accept on $x$, and hence the error is 0.

Now suppose $x \in L$. Define $N$ to be the diagonal matrix where $N_{ww} = 1$ if and only if $w \in B_x$. Otherwise, $N_{ww} = 0$ (and 0 everywhere off the diagonal). Let $q$ be some distribution on nodes of $G_n$. Then $||q \cdot N||_1$ is precisely the probability of drawing a bad string from $q$.

Let $P$ be the transition matrix corresponding to $G_n$. Then observe that $||qNP||_1$ is the probability of drawing a bad string from $q$, and that taking a random walk with starting distribution $q$ will also yield a bad string. More generally, $||q(NP)^kN||_1$ is the probability that the first $k + 1$ steps of the random walk yield a bad string.

We will use the following lemmas, which we will prove later, to arrive at our desired claim:
Lemma 2 For all $\pi$ such that $\sum \pi_i \leq 1$ and $\pi_i \geq 0$ for all $i$, $\|\pi PN\|_2 \leq \frac{1}{2} \|\pi\|_2$. 

Lemma 3 For all $x \in \mathbb{R}^m$, $\|x\|_1 \leq 2^{m/2} ||x||_2$. 

$A'$ starts the random walk with the uniform distribution $\pi_0$. The probability that after $k$ iterations, $A'$ does not accept is precisely $\|\pi_0(NP)^k N\|_1$. This is bounded from above by $\|\pi_0(PN)^k\|_1$, where we have ignored the first $N$ factor. This is because without the first $N$ terms we are now allowing the initial choice of random string to fall outside the bad set, which means there are more ways of arriving at the bad set after $k$ steps. Hence,

$$
Pr[A' \text{ does not accept}] \leq \|\pi_0(PN)^k\|_1 \\
\leq 2^{r/2} \|\pi_0(PN)^k\|_2 \\
\leq 2^{r/2} \frac{1}{5k} \|\pi_0\| \\
\leq \frac{1}{5^k}.
$$

Proof (Of Lemma 2). Since $G_n$ is $d$-regular, we have that $P$ is symmetric and hence diagonalizable. Let $\{v_i\}$ be the orthonormal basis in which $P$ is diagonal, such that $1 = \lambda_1 > |\lambda_2| \geq \cdots \geq |\lambda_2^r|$. Let $\pi = \sum \alpha_i v_i$. Then,

$$
\|\pi PN\|_2 = \left\| \alpha_1 v_1 PN + \sum_{i>2} \alpha_i v_i PN \right\|_2 \\
\leq \|\alpha_1 v_1 N\|_2 + \left\| \sum_{i>2} \alpha_i \lambda_i v_i N \right\|_2
$$

where the second line follows from the triangle inequality. We bound each term separately. Since $\lambda_1$, and $v_1 = \frac{1}{\sqrt{d}} (1,1,\ldots,1)$, we have that $\|\alpha_1 v_1 N\|_2 = \alpha_1 ||v_1 N||_2 \leq \alpha_1 \sqrt{1/100} \leq \frac{1}{100} \|\pi\|_2$.

For the second term, we note that $\left\| \sum \alpha_i \lambda_i v_i N \right\|_2 \leq |\lambda_2| \left\| \sum \alpha_i v_i N \right\|_2 \leq \frac{1}{10} \|\pi\|_2$, because multiplying by $N$ does not grow the length of a vector. Combining the two bounds yields the lemma.

Proof (Of Lemma 3). This follows from applying Cauchy-Schwarz: $(\sum x_i^2) = (\sum 1 \cdot x_i)^2 \leq (\sum 1^2) (\sum x_i^2) = 2^m (\sum x_i^2)$. 

This (temporarily) concludes the section on random walks on graphs. Next, we turn towards the topics of Fourier Analysis and Linearity Testing. We end this class with a prelude of program checking.

5 Program Checking and Linearity Testing

Suppose you can’t prove that a program $P$ correctly computes $f(x)$ on all inputs $x$, but you don’t care – you just want to verify that it’s correct on a specific input. Is it possible to create a program checker $C$ that will do this for you? We’d want a program that would tell us if $P(x)$ is correct or not. Of course, the checker itself might be faulty, but what we want is for the checker to be faulty in a way that’s independent of the way $P$ is faulty. A program that admits such a program checker is called checkable. What kind of programs are checkable?

One class of checkable programs are ones computing functions that are close to linear. Let’s see some definitions.
Definition 4 (Linear function) A function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is linear if and only if for all $x, y \in \{0, 1\}^n$, $f(x + y) = f(x) + f(y)$, where addition is performed component-wise modulo 2.

Definition 5 ($\epsilon$-close to linear) A function $g : \{0, 1\}^n \rightarrow \{0, 1\}$ is $\epsilon$-close to linear if and only if there exists a linear function $f$ such that $\Pr_x[g(x) = f(x)] \geq 1 - \epsilon$.

We describe one component of a program checker, called a self-corrector, for programs that compute linear functions. Suppose we have a program $P$ computing a function $g$ that is $\epsilon$-close to a linear function $f$, and we intended $P$ to compute $f$. We can correct $P$ so that it computes $f$ exactly.

- **Program Self-Corrector$(x)$**
  - For $O\left(\frac{1}{\epsilon}\right)$ iterations:
    * Pick $y \in \{0, 1\}^n$ uniformly at random.
    * Let $\alpha \leftarrow P(x + y) + P(y)$.
  - Output the majority of the $\alpha$'s.

Claim 6 With very high probability, Self-Corrector will compute $f$.

Proof Note that at each iteration, $y$ is chosen uniformly at random, so thus $x + y$ and $y$ are uniformly distributed. Thus, with probability at least $1 - 2\epsilon$, $P(x + y) = f(x + y)$ and $P(y) = f(y)$. By the linearity of $f$, $P(x + y) + P(y) = f(x + y) + f(y) = f(x)$. Since we’re picking $y$ independently with each iteration, the probability that the majority of the $\alpha$’s agree with $f(x)$ rapidly approaches 1. Thus, with high probability, Self-Corrector will output the correct value of $f(x)$. ■

Generally, program checkers will combine self correctors with program “testers”, which we still study next time.