6.842 Randomness and Computation October 11, 2017

Lecture 10

Lecturer: Ronitt Rubinfeld *Scribe: Tom Kolokotrones*

1 Space Bounded and Random Turing Machines and RL

1.1 Space Bounded and Random Turing Machines

When studying space-bounded computation, we must be somewhat careful about how we define a Turing Machine, particularly when the space bound is sublinear. In order to accommodate this (important) special case, we will use a multitape Turing Machine with several additional restrictions. The first tape is the input tape; it is read-only, but can otherwise be accessed normally. The final tape is the output tape; it is write-only so that the Turing Machine can produce arbitrarily long output, but cannot use the tape in order to circumvent the space limit. To make the output more elegant, we can also permit the Turing Machine to freely erase the output tape at any time, but this doesn't change the machine's power. The remaining tapes are the work tapes; they are read/write and behave exactly as normal Turing Machine tapes do. When we refer to the space complexity of a computation, we are referring only to the amount of work tape used.

We will also need to define a Randomized Turing Machine. Such a TM differs from the standard form in that it has access to a supply of random bits. One can imagine this source either as a sort of oracle that the TM can call at each step to generate a random symbol (typically 0 or 1), or as having access to an infinite tape full of random symbols, which the Turing Machine can read in one at a time. However, using that version, the head on the random tape can only move to the right and, each time the TM reads in a new random symbol, the tape also advances a single step, so that the only way to recall a previously used random bit, is to store it. These extra limitations are needed in space-bounded models in order to prevent the Turing Machine from being able to "store" bits outside of the work tape. Further, for a Randomized Turing Machine to be a decider, we require that it halt on every input, regardless of the randomness used.

1.2 Random Space Bounded Computation and RL

The question is, how much extra computation space do you need? We're only going to charge for the space that we actually get to use to do work, where we can read and write. So what can we keep there? We can keep pointers, we can copy the input or part of the input, we can do addition or subtraction or keep track of mins and maxes. We can do all kinds of things here. So, what we want to define now is the class **RL**, which is the class of problems for which you only need (randomized) logarithmic space.

Definition 1 RL — *A language $L \in$ **RL** iff there exists a Randomized Turing Machine which, given an input x of length n, always halts, uses at most $O(\log n)$ space, always rejects if $x \notin L$, and accepts with probability $\frac{1}{2}$ or greater if $x \in L$.*

Now, what can we keep in logarithmic space? Just numbers and pointers, that's it. We can't copy the whole input. We can hardly keep anything in there. So, what we're going to do today is randomized logspace computation.

Let's point out that dynamic programming does not tend to work in logspace, because, in dynamic programming, the idea is that you don't want to recompute things, that means you store it, but where would you store it? You only have logspace. Usually the algorithms that do it, for example in a graph problem, keep order n^2 at least, maybe $n^2 \log n$ bits, just to keep track of the best solution at the first level, the next, etc. Just think of memoization, where you're keeping track of a ton of stuff sometimes, n^2, n^3, and you need the best solution for each subproblem so you don't have to go recompute. So that's what dynamic programming does, it uses a ton of space, well, it's usually not exponential, it's polynomial, but we're in an an extreme place here, random logspace. What can we do with very little extra space? All we have is room for a few pointers, maybe some numbers, and a counter or two. However, what we're going to show is that **USTCON** $\in$ **RL**.

2 USTCON is in RL

2.1 A Randomized Algorithm for USTCON

Even though the typical ways to find paths, BFS, DFS, the ways that work in maybe the fastest time, need a lot of space, there is a way to do Undirected ST-Connectivity in logspace. You can think of those Roomba vacuum cleaners, you know those little guys, they don't really have to keep amazing maps. They can just keep a little bit of memory to figure out where they are, to get through the room. So, you can think of the vacuum cleaner, what would the vacuum cleaner do?

So, what would be the algorithm? Last time we talked about random walks on graphs, so you're probably guessing that we're going to use them, and, yes, you're right. You're thinking just the way I wanted you to think, because we are going to take a random walk on a graph. In fact, that's the whole algorithm. So, we are going to start at s. So, what's the Turing Machine going to do? It's going to scan the description until it gets to the row for s and then start a random walk at s. So, we only need to scan; we only need a pointer to go down this thing. We're going to scan down that input until we find the first place that we have the row written down for s and then what we're going to start doing is to take a random walk. We pick an out neighbor with uniform probability and then keep doing that. So, the algorithm is:

On input G, an undirected graph:

1. Start at s.
2. Take a random walk for cn^3 steps.
    (a) If ever we encounter t, output **YES**.
    (b) Otherwise output **NO**.

We're going to choose c such that cn^3 is at least something like 4 times the Cover Time. Remember that, for a graph G, the Cover Time, $\mathcal{C}(G) = \max_{s \in \mathcal{N}(G)} \mathrm{E}[\#$ steps to reach all nodes of G starting at $s]$. We showed that the max cover time is $O(n^3)$, so it takes time n^3 times some constant. Maybe it's 2, maybe it's 3, maybe it's 15. I want to take 4 times that many steps, so it's some other constant times n^3. So I'm taking 4 times as many steps as the expected cover time. So, first of all, what do we need to keep track of to implement this thing? What are my space needs? We need a counter to keep track of how many random walk steps: that's $O(\log n)$. We need pointers for scanning to pick our next step. We need to choose uniformly from each of the adjacent vertices. If we're at some node u, we need to figure out its degree d, so we need a counter for that and then need to toss a fair d-sided die (so that each face has a probability of d^{-1}) to determine which neighbor m of u to which to move. Then, we need to scan again through u's neighbors until we reach the m^{th} neighbor and then that will be the next node in the random walk.

So, our time is polynomial, but it's not linear, whereas other things for doing connectivity are linear, which is great, but this is at least polynomial. Each step we take requires that we toss enough coins to simulate a die. So, first, we need to figure out the degree of u, then we need to toss a die, then we need to scan through the neighbors of u until we find the m^{th} one. Each step will take linear time (in the number of nodes). Each step is $O(n)$, we need to do $O(n^3)$ steps, so this is $O(n^4)$ running time.

We need a counter for the number of steps, a counter for degree, a counter for which neighbor is chosen, and a pointer for scans, so this is not so bad. This whole thing is $O(\log n)$ space and the time used is $O(n^4)$.

2.2 A Little Complexity Theory

This is not really a complexity class, but let's just think for a second. Can we have something run in exponential time, but in logarithmic space? No, because eventually the configuration will repeat itself. This is easy to see in the deterministic case. A Deterministic Turing Machine using logarithmic space only has $2^{O(\log n)} = \text{poly}(n)$ configurations for its space. So, if it were to run for exponential time, a configuration would have to repeat, and, because the TM is deterministic, it would then loop forever. Therefore, a logspace decider can run for at most polynomial time.

This is also true for Randomized Turing Machines, by similar, but somewhat more subtle, reasoning. If a Randomized Turing Machine runs for more than $2^{O(\log n)} = \text{poly}(n)$ time, then, some configuration, say s, must repeat. Therefore, there exists some sequence of random bits r_{ss}, which, beginning at s will cause the TM to return to s again. But then, by beginning at s and repeating r_{ss} over and over, the RTM can be made to loop as many times as desired, and, in particular, can loop forever. Since a decider must halt on any input, regardless of the random bits generated, this cannot occur, so a randomized logspace decider can run for, at most, polynomial time. In fact, we can show that $\mathbf{RL} \subseteq \mathbf{SC}$, where **SC** is Steve's Class (named after Steven Cook) which consists of all languages that can be decided in polynomial time and polylog space [1]. A more recent result shows that $\mathbf{RL} \in \mathbf{L}^{3/2}$ [2].

Similar reasoning also applies to nondeterministic logspace deciders, since we need only decide those branches not containing loops, so, again, we only need $2^{O(\log n)} = \text{poly}(n)$ time (a limitation we can explicitly impose using a counter).

2.3 Proving the Correctness of Our Algorithm

We have our space needs, which are logarithmic, and our time needs, which are polynomial. But we haven't checked whether the algorithm works. Let's do that. Why does the algorithm work? If s and t are not connected on a graph, like if they're in different connected components, and we start walking at s, always picking a neighbor in the random walk process, will we ever get to t? No, there is no way we can get to t. So, if s and t are not connected, $\mathbf{P}(\mathbf{YES}) = 0$. Therefore we have one-sided error. If $x \notin \mathbf{USTCON}$, we will never have an error in this case, and will always give the right answer.

If s and t are connected, we would be wrong if we answered, no. This would happen if we started at s and took a random walk, and s and t are connected, but we did not see t in this random walk, even though we went four times the cover time. So, the probability that we answered, no, is, at most, the probability that we didn't cover the graph in four times the cover time of G many steps. What is the Cover Time? It is maximum over all starting points of the expected number of steps that we need to take in order to cover the graph, which is, at least as large as, the expected number of steps we need to take starting at s. So, starting at s, if we didn't see t in four times the expected time, what is the probability of that? Markov's Inequality tells us that the probability is at most $\frac{1}{4}$.

Therefore, we showed that, if s and t are not connected, that this thing is never going to give you the wrong answer, and, if s and t are connected, then the probability that we don't see that they are connected is at most $\frac{1}{4}$. If we wanted it to be at most $\frac{1}{10}$, we could run for ten times the cover time. Markov's Inequality is really the dumbest thing we can do. Instead we could do this a few times and then use independence and get even better results.

What we just showed is that S-T Connectivity on undirected graphs is in random logspace. It is actually in deterministic logspace, so we don't actually need random walks, which leads us to the question is $\mathbf{L} = \mathbf{RL}$? Hmmm, we don't know. So that's the big open question. This course is about when is randomness useful, when is it helpful? We don't even know the answer for **L** vs. **RL**. This is the big open question. We used to think that $\mathbf{USTCON} \in \mathbf{L}$ was the canonical problem and, if we could solve this, then we'd be done, but this is not true. Next week we'll show that $\mathbf{USTCON} \in \mathbf{L}$. It's not just in **RL**, it's in **L**. This random walk thing was the first use of cover time and random walks in complexity theory, which dates back to the early 80's. The **RL** vs. **L** thing was a big result in 2004, so it was a few years in-between, about twenty years in-between [3]. Then, everyone thought that was it, we're going to solve the problem fast, but that didn't happen. Our best current result is $\mathbf{RL} \subseteq \mathbf{L}^{3/2}$.

What we saw is an application for cover time. This is one reason that you would care about cover time. There are a lot of reasons that you would care about cover time, there are a lot of reasons that you would care about random walks, we're about to see another one. But, before we look at another reason that we care about random walks, we need to make some connections between linear algebra and random walks, because it's going to give us a nice way of talking about the next thing we care about in random walks. We want to know not just how long it takes to cover the graph, but how long it takes to get to the stationary distribution. We call that the mixing time. So, we take a random walk, when is it that we expect to be at our stationary distribution? If we're on a complete graph with self-loops, then, after one step, we're at a random node. So the mixing time there is 1; it's great.

3 Random Walks and Linear Algebra

3.1 Eigenvectors and Eigenvalues

Definition 2 *Eigenvectors and Eigenvalues — Given a linear operator $T : V \to V$, where V is a vector field over F, if there exists some nonzero $v \in V$ and some $\lambda \in F$ such that $vT = \lambda v$, then v is a (left)* eigenvector *of T and λ is its corresponding* eigenvalue*.*

Definition 3 *ℓ^p norm — For $p \geq 1$ the ℓ^p* norm *of a vector $v \in V$ is given by $\|v\|_p = (\sum_i v_i^p)^{\frac{1}{p}}$.*

Theorem 1 *Given a transition matrix P for an undirected graph (which is stochastic, real, and symmetric, and, thus, doubly stochastic), there exists a (not necessarily unique) orthonormal basis of eigenvectors $v_1, \ldots, v_n$ (so that $v_i \cdot v_j = \sum_k v_{i,k} v_{j,k} = \delta_{ij}$), with corresponding eigenvalues $1 = \lambda_1, \lambda_2, \ldots, \lambda_n$ ordered in decreasing absolute value (so that $1 = |\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$) and $v_1 = n^{-\frac{1}{2}}(1, \ldots, 1)$.*

Using this choice of v_i's, v_1 is a scaled version of the stationary distribution, with $\|v_1\|_2 = 1$, which will be convenient for our purposes. We could have a chosen a different normalization for v_1, for example $\|v_1\|_1 = 1$, so that $v_1 = \pi$. If $\{v_i,\}_{i=1}^n$ are (left) eigenvectors of P with corresponding eigenvalues $\{\lambda_i\}_{i=1}^n$, and $\alpha \in F$, we have the following facts:

1. αP has (left) eigenvectors $\{v_i,\}_{i=1}^n$ with corresponding eigenvalues $\{\alpha\lambda_i\}_{i=1}^n$.
2. $\frac{P+I}{2}$ has (left) eigenvectors $\{v_i,\}_{i=1}^n$ with corresponding eigenvalues $\{\frac{\lambda_i+1}{2}\}_{i=1}^n$.
3. P^k has (left) eigenvectors $\{v_i,\}_{i=1}^n$ with corresponding eigenvalues $\{\lambda_i^k\}_{i=1}^n$.
4. For $\alpha \neq 0$, $\{\alpha v_i,\}_{i=1}^n$ are (left) eigenvectors of P with corresponding eigenvalues $\{\lambda_i\}_{i=1}^n$.
5. If P is stochastic then $\forall_i |\lambda_i| \leq 1$.

Proof

1. $v_i(\alpha P) = v_i \alpha P = \alpha v_i P = \alpha \lambda_i v_i = (\alpha\lambda_i) v_i$

2. $v_i \frac{P+I}{2} = \frac{1}{2}(v_i P + v_i) = \frac{1}{2}(\lambda_i v_i + v_i) = \frac{\lambda_i + 1}{2} v_i$

3. $v_i P^k = v_i P P^{k-1} = \lambda_i v_i P^{k-1} = \cdots = \lambda_i^k v_i$

4. $(\alpha v_i)P = \alpha v_i P = \alpha \lambda_i v_i = \lambda_i(\alpha v_i)$

5. $\lambda_i v_{i,j} = (\lambda_i v_i)_j = (v_i P)_j = \sum_k v_{i,k} P_{kj} \leq \sum_k \max_j(v_{i,j}) P_{kj} = \max_j(v_{i,j}) \sum_k P_{kj} = \max_j(v_{i,j})$. Where we used $P_{ij} \geq 0$ in the inequality and $\sum_k P_{kj} = 1$ in the final equality. Correspondingly, we also have $\lambda_i v_{i,j} \geq \min_j(v_{i,j})$. Therefore, $\min_j(v_{i,j}) \leq \lambda_i v_{i,j} \leq \max_j(v_{i,j})$. If v_i is an eigenvector of P, $v_i \neq 0$ and $-v_i$ is also an eigenvector of P with the same eigenvalue λ_i, so, we may assume, without loss of generality, that $\max_j(v_{i,j}) \geq \max_j(-v_{i,j}) = -\min_j(v_{i,j})$, which also implies $\max_j(v_{i,j}) > 0$, since $v_i \neq 0$. Then, $-\max_j(v_{i,j}) \leq \lambda_i v_{i,j} \leq \max_j(v_{i,j})$ so $-\max_j(v_{i,j}) \leq \lambda_i \max_j(v_{i,j}) \leq \max_j(v_{i,j})$ and $|\lambda_i| \leq 1$.

■

We also note the following important fact:

Fact 2 *If $\{v_i\}_{i=1}^n$ is a basis for V, then, for any $v \in V$, there exist $\alpha_i \in F$ such that $v = \sum_i \alpha_i v_i$. Further, if $\{v_i\}_{i=1}^n$ is an orthonormal basis then $\alpha_i = v \cdot v_i$, $v = \sum_i (v \cdot v_i) v_i$, and $\|v\|_2 = \left(\sum_{i,j} \alpha_i \alpha_j (v_i \cdot v_j)\right)^{\frac{1}{2}} = \left(\sum_{i,j} \alpha_i \alpha_j \delta_{ij}\right)^{\frac{1}{2}} = \left(\sum_i \alpha_i^2\right)^{\frac{1}{2}}$.*

3.2 Random Walks and Mixing Times

Since we have an orthonormal eigenbasis $\{v_i\}_{i=1}^n$, it means that we can take any vector and write it as a linear combination of the eigenvectors. This will help us because it will allow us to talk about what happens to a vector when we apply P to it a certain number of times. Say we take t steps, then we apply P^t, what happens to our vector?

What we want to talk about now, is the mixing time of a random walk. What we care about right now is how long does it take us to reach the stationary distribution when we take a random walk and that's the mixing time. We don't expect to actually reach the stationary state, we only want to reach an approximation of it.

Definition 4 Mixing Time — *Given an $\epsilon > 0$, the* mixing time, *$t(\epsilon)$ of a Markov Chain P, with stationary distribution π is the minimum t such that for any starting distribution π_0, $\|\pi - \pi_0 P^t\|_1 < \epsilon$.*

So, given some $\epsilon > 0$, we want to know at what time we get ϵ close to π in ℓ^1 norm, how many steps we need to take. We look at the worst possible starting distribution and at that worst possible one, we take t steps and ask when π_0, $\|\pi - \pi_0 P^t\|_1 < \epsilon$.

Definition 5 Rapid Mixing — *We say a Markov Chain P is* Rapidly Mixing *if $t(\epsilon)$ is polynomial in the number of nodes, n, and ϵ^{-1}.*

This is crazy. This is not what we were talking about before. This is much faster. Remember that we said in the complete graph you mix in one step? That's rapid mixing. But remember the lollipop graph when the cover time was $O(n^3)$? That's not rapid mixing. You can make a lollipop graph, or something like a lollipop graph, where everybody should be pretty much uniform; you can make it so it looks like everyone is pretty much degree 2 and so everything should be close to uniform, so it means that you pretty much need to see all the nodes in the graph. That is not rapid mixing. You need something polynomial in the size of the graph, not polynomial in log the number of nodes, but polynomial in the number of nodes. Rapid mixing is so much faster, it's like exponentially faster than in the lollipop graph, or the line, or any of those other graphs we saw last time, except for the complete graph.

These rapid mixing graphs, we are going to get these out of a hat, people are going to give these to us and we are going to use them. Now here's the cool thing, a random graph is rapid mixing, even a random 3-regular graph is rapid mixing. So there's a lot of rapid mixing graphs out there. When I say a random graph, I mean that if I pick all the edges at random, it's very likely to be rapid mixing. What does this mean? It means that most graphs are rapid mixing. However, finding a specific graph that is rapid mixing, is not always so simple. People have spent a lot of time on finding the best parameters, but a random one does work. Most of them are good; finding an explicit one with all the right properties is a little tougher, maybe sometimes a lot tougher, but that's all right. This class is a theory class, so we can just assume it exists.

3.3 Convergence to the Stationary Distribution

So, here's the coolest thing. Given some transition matrix P, for an undirected, nonbipartate, d-regular, connected graph and a starting distribution π_0, whatever it is, if you take t steps, then how far are you from the stationary distribution π? Remember that, for such a graph, the stationary state $\pi = n^{-1}(1, \ldots, 1)$, since an undirected graph has a symmetric transition matrix and so is doubly stochastic. Actually, what we're going to be interested is not the ℓ^1-norm, which we used to define mixing time, but rather the ℓ^2-norm. Recall that, for an n-dimensional vector space V, for all $v \in V$, the inequality $\|v\|_2 \leq \|v\|_1 \leq n^{\frac{1}{2}}\|v\|_2$ holds; we'll use this later. Our theorem is:

Theorem 3 *If P is the transition matrix of an undirected, nonbipartate, d-regular, connected graph and π_0 is any starting distribution, then $\|\pi_0 P^t - \pi\|_2 \leq |\lambda_2|^t$.*

Why is this interesting? Because λ_2 will be strictly less than 1. If λ_2 is very close to 1, that's not good, but if λ_2 is like .9, if we can get a graph where λ_2 is .9, then the ℓ^2 distance between the starting and stationary state will drop exponentially. If $\lambda_2 = 1 - n^{-1}$, then it's not that good yet, but it will be if we take a few more steps. So, it means that we need to take a number of steps that depends on the separation of λ_2 from 1. Now, people will often talk about the separation of λ_1 and λ_2 instead of the separation of λ_2 from 1. We happened to fix $\lambda_1 = 1$, so for us, these are the same thing because we set it this way, but when you hear other people talk it, they'll talk about the distance between the top two eigenvalues. Notice that if, $\lambda_2 = 1$, like if we had a disconnected graph and had another eigenvalue of size 1, then this tells us nothing, since 1^t is still one. The value is not going down at all as we take more steps, which we would expect, because in a disconnected graph, or a bipartate graph, things can be pretty crazy. But when things are nice, when $\lambda_2 < 1$, the ℓ^2 distance goes down exponentially. So that's cool, but even cooler is how we prove this.

So, we have our orthonormal eigenbasis $\{v_i\}_{i=1}^n$ and we've started at π_0. Using the orthonormal basis, we can write $\pi_0 = \sum_i \alpha_i v_i$. We're not doing anything weird, we're just using the fact that the v_i's form an orthonormal basis so we can write any vector, including π_0, as a linear combination of them. If we take π_0 and we multiply by P^t, what do we get? Well, we get $\pi_0 P^t = \sum_i \alpha_i P^t v_i = \sum_i \alpha_i \lambda_i^t v_i$. Do you see what we did here? This is a really cool trick. We took this thing, we wrote it in this new basis, and then, when we apply P^t, we can look at what happens to each component. Now, one of them has $\lambda_i = 1$, the first one, so $\pi_0 P^t = \sum_i \alpha_i \lambda_i^t v_i = \alpha_1 v_1 + \sum_{i=2}^n \alpha_i \lambda_i^t v_i = \pi + \sum_{i=2}^n \alpha_i \lambda_i^t v_i$, since $\lambda_1 = 1$ and $\alpha_1 v_1 = \pi$, since, as $t \to \infty$ all the other terms go to 0. Now, what do we have here? We want to get this in terms of λ_2. What we want to say is this: the λ_i^t's are dropping, plus/minus, we don't know yet, but in absolute value they're dropping, because λ_2 is the second biggest eigenvalue, so all the eigenvalues after λ_2 have absolute value at most $|\lambda_2|$ because we sorted them. We want to say that we can just replace them by λ_2^t modulo some absolute value issue that we'll talk about in a minute. Then $\pi_0 P^t - \pi \leq \sum_{i=2}^n \alpha_i \lambda_2^t v_i$, which will give us the inequality that we want. More formally,

$$
\begin{aligned}
\left\|\pi_0 P^t - \pi\right\|_2 &= \left\|\sum_i \alpha_i \lambda_i^t v_i - \pi\right\|_2 = \left\|\alpha_1 v_1 + \sum_{i=2}^n \alpha_i \lambda_i^t v_i - \pi\right\|_2 = \left\|\pi + \sum_{i=2}^n \alpha_i \lambda_i^t v_i - \pi\right\|_2 \\
&= \left\|\sum_{i=2}^n \alpha_i \lambda_i^t v_i\right\|_2 = \left(\sum_{i=2}^n \alpha_i^2 \lambda_i^{2t}\right)^{\frac{1}{2}} \le \left(\sum_{i=2}^n \alpha_i^2 \lambda_2^{2t}\right)^{\frac{1}{2}} = |\lambda_2|^t \left(\sum_{i=2}^n \alpha_i^2\right)^{\frac{1}{2}} \\
&\le |\lambda_2|^t \left(\sum_{i=1}^n \alpha_i^2\right)^{\frac{1}{2}} = |\lambda_2|^t \|\pi_0\|_2 \le |\lambda_2|^t \|\pi_0\|_1 = |\lambda_2|^t \cdot 1 \\
&= |\lambda_2|^t
\end{aligned}
$$

so $\left\|\pi_0 P^t - \pi\right\|_2 \le |\lambda_2|^t$, proving the theorem.

So, what do we do in our proof? We take the starting vector π_0, which is arbitrary, but we can write it in terms of the orthonormal eigenbasis $\{v_i\}_{i=1}^n$. Then, when we apply the transition matrix, P, we can apply it to each component separately. All of the components, except for the first one, are all dropping down to zero because their λ_i's are strictly less than one. $\lambda_2 < 1$ is the second largest, but all the others are even smaller, so those components are dropping down very quickly. So, when we apply P^t, the λ_i's are dropping down really fast, they become zero very quickly. So, everybody but the first vector are all going down to nothing, very fast, exponentially fast, but the first guy, that's what we're tending to. That's our stationary distribution, that's what we're going to.

Now, what's the issue? λ_2 might be close to one so it might take us some time until we get to the point before we start dropping down exponentially. λ_2 might be something like $1 - n^{-1}$ or something really close to one, so that might be an issue for us. But, if λ_2 is a constant bounded away from one, it will drop down exponentially and everything's great. All the rest of the terms are just going right to zero, very quickly, and that's the cool thing about this. The linear algebra perspective gave us another way to view these vectors and we could see what's happening by applying the transition matrix. All this arbitrary stuff is going away and we're just going straight to the stationary distribution. It doesn't matter where we started, everything is going away. So that's the cool thing here and why we like linear algebra.

We're going to use this next time to save randomness, in a different way from pairwise independent hashing, which will be slightly better in some ways than pairwise independence.

References

[1] Nisan, Noam (1992),“RL ? S”, Proceedings of the 24th ACM Symposium on Theory of computing (STOC ’92), Victoria, British Columbia, Canada, pp. 619623.

[2] Vadhan, Salil. Pseudorandomness. Foundations and Trends in Theoretical Computer Science Vol. 7, Nos. 1-3 (2011) 1-336.

[3] O. Reingold and L. Trevisan and S. Vadhan. Pseudorandom walks in biregular graphs and the RL vs. L problem, ECCC TR05-022, 2004.