Fooling machines that have limited computational power

Karteek Sreenivasaiah

22nd July 2020

Randomness

Is there randomness in the universe?

Randomness

Is there randomness in the universe?

We do not know.

Randomness

Is there randomness in the universe?

We do not know.

But anyhoo, the answer does not affect purely theoretical areas like:

- Probability Theory
- Probabilistic Method
- Discrete mathematics, combinatorics

Randomness

Is there randomness in the universe?

We do not know.

But anyhoo, the answer does not affect purely theoretical areas like:

- Probability Theory
- Probabilistic Method
- Discrete mathematics, combinatorics

Why is this question interesting to mathematics?

Randomness

An area of computer science that needs the answer to be Yes is:

Randomized Algorithms

Randomness

An area of computer science that needs the answer to be Yes is:

Randomized Algorithms

- One of the most elegant areas of computer science.
- Almost always use lesser resources than deterministic counterparts.
- Many times there are no deterministic counterparts.
- Randomized algorithms are used widely in practice.

Hence the question of the existence of randomness is very important.

"Toss a coin dude, it'll look random lol"

– some dude on the internet, maybe

Coin Toss

Tossing a coin identically will give identical outcomes.

So why do coin tosses appear random to us?!

Coin Toss

Tossing a coin identically will give identical outcomes.

So why do coin tosses appear random to us?!

If we could sense the following at the instant of tossing:

- The force we apply
- The point of contact
- Angle of attack, bla bla bla

Then the outcome is simply a deterministic function.

Coin Toss

Tossing a coin identically will give identical outcomes.

So why do coin tosses appear random to us?!

If we could sense the following at the instant of tossing:

- The force we apply
- The point of contact
- Angle of attack, bla bla bla

Then the outcome is simply a deterministic function.

The function could be very hard to compute before the coin lands!

Randomness

Could it be true that limited computational power makes events look completely random even if they are not?

Randomness

Could it be true that limited computational power makes events look completely random even if they are not?

Nisan '92 shows that indeed this is true, and proves it formally!

Nisan '92

COMBINATORICA
Akadémiai Kiadó – Springer-Verlag

COMBINATORICA **12** (4) (1992) 449–461

PSEUDORANDOM GENERATORS FOR SPACE-BOUNDED COMPUTATION

NOAM NISAN*

Received December 3, 1989
Revised June 16, 1992

Pseudorandom generators are constructed which convert $O(S \log R)$ truly random bits to R bits that appear random to any algorithm that runs in $SPACE(S)$. In particular, any randomized polynomial time algorithm that runs in space S can be simulated using only $O(S \log n)$ random bits. An application of these generators is an explicit construction of universal traversal sequences (for arbitrary graphs) of length $n^{O(\log n)}$.

The generators constructed are technically stronger than just appearing random to space-bounded machines, and have several other applications. In particular, applications are given for "deterministic amplification" (i.e. reducing the probability of error of randomized algorithms), as well as generalizations of it.

This talk borrows ideas and some notation from the excellent lecture notes by Ryan O' Donell. `https://www.cs.cmu.edu/~odonnell/complexity/docs/lecture16.pdf`

Preliminaries

The Problem:

There is a randomized algorithm $\mathcal{A}$ that:

- uses R independent random bits drawn from U_R.
- uses space $O(s)$.

Preliminaries

The Problem:

There is a randomized algorithm $\mathcal{A}$ that:

- uses R independent random bits drawn from U_R.
- uses space $O(s)$.

We would like to:

- Draw only $O(s \log R)$ random bits from uniform. Call this x.
- Generate a string y of length R using x deterministically.
- Feed y to $\mathcal{A}$ as the "random" bits.
- Output Yes or No with probabilities similar to that of $\mathcal{A}$.

Think of $s \in O(\log n)$. Then, $O(s \log R) \in O(log^2 n)$.

Definition

A function $G : \{0,1\}^n \to \{0,1\}^m$ ϵ-fools a randomized algorithm A that uses m bits of randomness if for all inputs x

$$\left| \Pr_{r \sim U_m}[A(x,r) \text{ accepts}] - \Pr_{y \sim U_n}[A(x,G(y)) \text{ accepts}] \right| \le \epsilon$$

Definition

A function $G : \{0,1\}^n \to \{0,1\}^m$ ϵ-fools a randomized algorithm A that uses m bits of randomness if for all inputs x

$$\left| \Pr_{r \sim U_m}[A(x,r) \text{ accepts}] - \Pr_{y \sim U_n}[A(x, G(y)) \text{ accepts}] \right| \leq \epsilon$$

We should think of $n << m$.
i.e., G takes a small string of length n from the uniform distribution and stretches it to a length m string that *looks random* to the algorithm A.

Such a function is called a pseudorandom generator.

Our goal is to construct a pseudorandom generator that can fool every space $O(s)$ algorithm.

Preliminaries

Some Assumptions:

- The input x is given to us.
- The algorithm $\mathcal{A}$ uses randomness in blocks of k bits.
- The TM A corresponding to $\mathcal{A}$ has a unique accepting configuration.

A *configuration* of a TM typically looks like this:

$$0110001101010\ q_4\ 0011111010\sqcup$$

Preliminaries

Some Assumptions:

- The input x is given to us.
- The algorithm $\mathcal{A}$ uses randomness in blocks of k bits.
- The TM A corresponding to $\mathcal{A}$ has a unique accepting configuration.

A *configuration* of a TM typically looks like this:

$$0110001101010\ q_4\ 0011111010\sqcup$$

Fact: A TM that uses space s

- has at most $2^{O(s)}$ configurations.
- has running time at most $2^{O(s)}$.

Preliminaries

From the given TM A and input x, we construct the following state machine:

- State/Vertex set V is the set of all $m = 2^{O(s)}$ possible configurations.
- The start state is c_0 and accepting state is c_{acc}.
- c_{acc} has a self loop. No outgoing edges.
- Transitions are labelled by strings in $\{0, 1\}^k$.

The edges correspond to transitions from a configuration u to v after reading k random bits.

Preliminaries

From the given TM A and input x, we construct the following state machine:

- State/Vertex set V is the set of all $m = 2^{O(s)}$ possible configurations.
- The start state is c_0 and accepting state is c_{acc}.
- c_{acc} has a self loop. No outgoing edges.
- Transitions are labelled by strings in $\{0, 1\}^k$.

For every pair of vertices $u, v \in V$,

$$(u, v) \in E \text{ with label } t \in \{0,1\}^k$$
$$\Updownarrow$$
$$A \text{ goes from } u \text{ to } v \text{ after reading } t \text{ as the random string.}$$

Preliminaries

From the given TM A and input x, we construct the following state machine:

- State/Vertex set V is the set of all $m = 2^{O(s)}$ possible configurations.
- The start state is c_0 and accepting state is c_{acc}.
- c_{acc} has a self loop. No outgoing edges.
- Transitions are labelled by strings in $\{0, 1\}^k$.

For every pair of vertices $u, v \in V$,

$$(u, v) \in E \text{ with label } t \in \{0, 1\}^k$$
$$\Updownarrow$$
$$A \text{ goes from } u \text{ to } v \text{ after reading } t \text{ as the random string.}$$

Observation: Every state (except c_{acc}) has 2^k transitions going out. Denote the above state machine as an *(m,k)-automaton.*

Preliminaries

An (m, k)-automaton D will have a transition function that looks like:

$$\delta : V \times \{0, 1\}^k \to V$$

$\delta(u; x) = v \iff D$ goes from state u to v with x as the random string

Preliminaries

An (m, k)-automaton D will have a transition function that looks like:

$$\delta : V \times \{0, 1\}^k \to V$$

$\delta(u; x) = v \iff D$ goes from state u to v with x as the random string

Let M be the *transition* matrix of the state machine defined as:

$$M[u, v] = \Pr_x[\delta(u; x) = v]$$

Preliminaries

An (m, k)-automaton D will have a transition function that looks like:

$$\delta : V \times \{0, 1\}^k \to V$$

$\delta(u; x) = v \iff D$ goes from state u to v with x as the random string

Let M be the *transition* matrix of the state machine defined as:

$$M[u, v] = \Pr_x[\delta(u; x) = v]$$

Let the running time be $t = 2^{cs}$.

Our goal is to output "Yes" with probability close to $M^t[c_0, c_{\text{acc}}]$.

(Think of powering adjacency matrices for unweighted graphs)

Main Idea

A fast way to power matrices is via repeated squaring.

$$M \longrightarrow M^2 \longrightarrow M^4 \longrightarrow \cdots \longrightarrow M^t$$

Main Idea

A fast way to power matrices is via repeated squaring.

$$M \longrightarrow M^2 \longrightarrow M^4 \longrightarrow \cdots \longrightarrow M^t$$

Let's look at just M^2 for now.
How could we output "Yes" with probability close to $M^2[u, v]$?

Main Idea

A fast way to power matrices is via repeated squaring.

$$M \longrightarrow M^2 \longrightarrow M^4 \longrightarrow \cdots \longrightarrow M^t$$

Let's look at just M^2 for now.
How could we output "Yes" with probability close to $M^2[u, v]$?

Naïve way:

- Pick $x_1, x_2 \in \{0, 1\}^k$ uniformly and independently at random.
- Follow the path in the automaton graph starting from u labelled x_1 and x_2.
- Output "Yes" if we land at v.

i.e., Output "Yes" if $\delta(\delta(u; x_1); x_2) = v$.

By definitions, we would output "Yes" with probability $M^2[u, v]$.

Can we use fewer random bits to get a similar effect?

Main Idea

Nisan's idea:

- to pick string x_1 at random.
- Generate $x_2 = h(x_1)$ where h is a hash function picked from a pairwise independent hash family.

Main Idea

Nisan's idea:

- to pick string x_1 at random.
- Generate $x_2 = h(x_1)$ where h is a hash function picked from a pairwise independent hash family.

Short detour into pairwise independent hash families...

Universal Hash Families

Definition:

A family H of functions from $h : \{0,1\}^n \to \{0,1\}^m$ is a pairwise independent hash family if for all $x_1, x_2 \in \{0,1\}^n$, $x_1 \neq x_2$, and $y_1, y_2 \in \{0,1\}^m$, we have:

$$\Pr_h[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{2^{2m}}$$

Universal Hash Families

Definition:

A family H of functions from $h : \{0,1\}^n \to \{0,1\}^m$ is a pairwise independent hash family if for all $x_1, x_2 \in \{0,1\}^n$, $x_1 \neq x_2$, and $y_1, y_2 \in \{0,1\}^m$, we have:

$$\Pr_h[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{2^{2m}}$$

Definition:

Let $A, B \subseteq \{0,1\}^k$ and $h : \{0,1\}^k \to \{0,1\}^k$. Let $\alpha = |A|/2^k$ and $\beta = |B|/2^k$. The function h is "τ-independent for (A, B)" if

$$\left| \Pr_x[x \in A \wedge h(x) \in B] - \alpha\beta \right| < \tau$$

Universal Hash Families

Definition:
A family H of functions from $h : \{0,1\}^n \to \{0,1\}^m$ is a pairwise independent hash family if for all $x_1, x_2 \in \{0,1\}^n$, $x_1 \neq x_2$, and $y_1, y_2 \in \{0,1\}^m$, we have:

$$\Pr_h[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{2^{2m}}$$

Definition:
Let $A, B \subseteq \{0,1\}^k$ and $h : \{0,1\}^k \to \{0,1\}^k$. Let $\alpha = |A|/2^k$ and $\beta = |B|/2^k$. The function h is "τ-independent for (A, B)" if

$$\left|\Pr_x[x \in A \wedge h(x) \in B] - \alpha\beta\right| < \tau$$

Fact: If h is chosen at random from a pairwise independent hash family, then:

$$\Pr_h[h \text{ is } \textbf{not } \tau\text{-independent for } (A, B)] \leq \frac{1}{\tau^2 2^k}$$

Main Idea

Define shorthand $\delta^2(u; x_1, x_2) = \delta(\delta(u; x_1); x_2)$. Then we have:

$$\mathcal{M}^2[u, v] = \Pr_{x_1, x_2}[\delta^2(u; x_1, x_2) = v]$$

Main Idea

Define shorthand $\delta^2(u; x_1, x_2) = \delta(\delta(u; x_1); x_2)$. Then we have:

$$M^2[u, v] = \Pr_{x_1, x_2}[\delta^2(u; x_1, x_2) = v]$$

Define M_h to be the following matrix:

$$M_h[u, v] = \Pr_{x_1}[\delta^2(u; x_1, h(x_1)) = v]$$

Main Idea

Define shorthand $\delta^2(u; x_1, x_2) = \delta(\delta(u; x_1); x_2)$. Then we have:

$$M^2[u, v] = \Pr_{x_1, x_2}[\delta^2(u; x_1, x_2) = v]$$

Define M_h to be the following matrix:

$$M_h[u, v] = \Pr_{x_1}[\delta^2(u; x_1, h(x_1)) = v]$$

How different are M^2 and M_h?

Main Idea

Lemma: Let D be an $(m,k)-$automaton, and M it's transition matrix. Then:

$$\Pr_{h \sim H_k}[\|M^2 - M_h\|_\infty \geq \epsilon] \leq \frac{m^7}{\epsilon^2 2^k}$$

where $\|M\|_\infty$ denotes the the largest row sum of abs values in the matrix.

Main Idea

Proof: Fix an entry u, v, and assume h has been picked from a pairwise independent hash family. Then we have:

$$
\begin{aligned}
&|M[u,v] - M_h[u,v]| \\
&= \left| \Pr_{x_1,x_2}[\delta^2(u; x_1, x_2) = v] - \Pr_{x}[\delta^2(u; x, h(x)) = v] \right| \\
&= \left| \sum_{w=1}^{m} \Pr_{x_1,x_2}[\delta(u; x_1) = w \wedge \delta(w; x_2) = v] - \sum_{w=1}^{m} \Pr_{x}[\delta(u; x) = w \wedge \delta(w; h(x)) = v] \right| \\
&\leq \sum_{w=1}^{m} \left| \Pr_{x_1,x_2}[\delta(u; x_1) = w \wedge \delta(w; x_2) = v] - \Pr_{x}[\delta(u; x) = w \wedge \delta(w; h(x)) = v] \right| \\
&\leq \sum_{w=1}^{m} \left| \Pr_{x_1}[\delta(u; x_1) = w] \Pr_{x_2}[\delta(w; x_2) = v] - \Pr_{x}[\delta(u; x) = w \wedge \delta(w; h(x)) = v] \right|
\end{aligned}
$$

Main Idea

Proof: Fix an entry u, v, and assume h has been picked from a pairwise independent hash family. Then we have:

$$|M[u,v] - M_h[u,v]|$$

$$= \left| \Pr_{x_1,x_2}[\delta^2(u;x_1,x_2) = v] - \Pr_{x}[\delta^2(u;x,h(x)) = v] \right|$$

$$= \left| \sum_{w=1}^{m} \Pr_{x_1,x_2}[\delta(u;x_1) = w \wedge \delta(w;x_2) = v] - \sum_{w=1}^{m} \Pr_{x}[\delta(u;x) = w \wedge \delta(w;h(x)) = v] \right|$$

$$\leq \sum_{w=1}^{m} \left| \Pr_{x_1,x_2}[\delta(u;x_1) = w \wedge \delta(w;x_2) = v] - \Pr_{x}[\delta(u;x) = w \wedge \delta(w;h(x)) = v] \right|$$

$$\leq \sum_{w=1}^{m} \left| \Pr_{x_1}[\delta(u;x_1) = w] \Pr_{x_2}[\delta(w;x_2) = v] - \Pr_{x}[\delta(u;x) = w \wedge \delta(w;h(x)) = v] \right|$$

Define $A_{u,w} = \{x \mid \delta(u;x) = w\}$ and $B_{w,v} = \{x \mid \delta(w;x) = v\}$.

Main Idea

Suppose h is $\tau = (\epsilon/m^2)$-indep *for every* $(A_{u,w}, B_{w,v})$.
Then by definition of τ-independence, and union bound, we get:

$$|M[u,v] - M_h[u,v]| \leq m \cdot (\epsilon/m^2) = \frac{\epsilon}{m}$$

From property of hash family:

$$\Pr_h[h \text{ is not } \tau\text{-independent for } A_{u,w}, B_{w,v})] \leq \frac{1}{\tau^2 2^k} = m^4/\epsilon^2$$

Union bound over all u, w, v to get:

$$\Pr_h[h \text{ is bad}] \leq m^3 \cdot \frac{m^4}{\epsilon^2} = \frac{m^7}{\epsilon^2 2^k}$$

□

Main Idea

In picking $x \in \{0,1\}^k$ and an $h \in H_k$, did we really save a lot?

- Hash families with linear space descriptions are known.
- So choosing $h \in H_k$ needs only $O(k)$ bits of randomness.
- We could have chosen x_2 directly instead?!

Main Idea

In picking $x \in \{0,1\}^k$ and an $h \in H_k$, did we really save a lot?

- Hash families with linear space descriptions are known.
- So choosing $h \in H_k$ needs only $O(k)$ bits of randomness.
- We could have chosen x_2 directly instead?!

The idea of using hash functions scales extremely well:
Computing $M^4[u, v]$:

- We pick $x \in \{0,1\}^k$ and only two hash functions h_1, h_2.
- The strings we generate are x, $h_1(x)$, $h_2(x)$ and $h_1(h_2(x))$.

Main Idea

In picking $x \in \{0,1\}^k$ and an $h \in H_k$, did we really save a lot?

- Hash families with linear space descriptions are known.
- So choosing $h \in H_k$ needs only $O(k)$ bits of randomness.
- We could have chosen x_2 directly instead?!

The idea of using hash functions scales extremely well:

Computing $M^4[u, v]$:

- We pick $x \in \{0,1\}^k$ and only two hash functions h_1, h_2.
- The strings we generate are x, $h_1(x)$, $h_2(x)$ and $h_1(h_2(x))$.

Computing $M^8[u, v]$:

- We pick $x \in \{0,1\}^k$ and only three hash functions h_1, h_2, h_3.
- The strings we generate are:
  $x, h_1(x), h_2(x), h_1h_2(x), h_3(x), h_1h_3(x), h_2h_3(x), h_1h_2h_3(x)$.

Main Idea

In general, to compute M^{2^s}, we will use s many hash functions. And we will still be very close to M^{2^s}:

Theorem

$$\Pr_{h_1,h_2,\ldots,h_s \sim H_k}\left[\left\|M^{2^s} - M_{h_1,h_2,\ldots,h_s}\right\| > (2^s - 1)\epsilon\right] \leq s\frac{m^7}{\epsilon^2 2^k}$$

Main Idea

In general, to compute M^{2^s}, we will use s many hash functions. And we will still be very close to M^{2^s}:

Theorem

$$\Pr_{h_1,h_2,\ldots,h_s \sim H_k}\left[\left\|M^{2^s} - M_{h_1,h_2,\ldots,h_s}\right\| > (2^s - 1)\epsilon\right] \leq s\frac{m^7}{\epsilon^2 2^k}$$

Let's calculate the number of pure random bits used for the general case of estimating M^{2^s}:

- Picking $x \in \{0, 1\}^k$ needs k bits of randomness
- Picking $h_1, \ldots h_s$ needs $O(sk)$ bits of randomness.

Think of $s \in O(\log n)$, and choose $k \in O(s)$.
This gives number of random bits needed as $O(s^2) \in O(log^2 n)$.

Main Idea

In general, to compute M^{2^s}, we will use s many hash functions. And we will still be very close to M^{2^s}:

Theorem

$$\Pr_{h_1,h_2,\ldots,h_s \sim H_k}\left[\left\|M^{2^s} - M_{h_1,h_2,\ldots,h_s}\right\| > (2^s - 1)\epsilon\right] \leq s\frac{m^7}{\epsilon^2 2^k}$$

Let's calculate the number of pure random bits used for the general case of estimating M^{2^s}:

- Picking $x \in \{0,1\}^k$ needs k bits of randomness
- Picking $h_1, \ldots h_s$ needs $O(sk)$ bits of randomness.

Think of $s \in O(\log n)$, and choose $k \in O(s)$.
This gives number of random bits needed as $O(s^2) \in O(log^2 n)$.

Choosing $\epsilon \in 1/2^{O(s)}$ works for the bounds.
Please see the paper for the exact choices!

Pseudorandom Generator

The generator from the paper is defined recursively:

$$G_0(x) = x$$

$$G_x(x, h_1, \ldots, h_k) = G_{k-1}(x, h_1, \ldots, h_{k-1}) \circ G_{k-1}(h_k(x), h_1, \ldots, h_{k-1})$$

Pseudorandom Generator

The generator from the paper is defined recursively:

$$G_0(x) = x$$

$$G_x(x, h_1, \ldots, h_k) = G_{k-1}(x, h_1, \ldots, h_{k-1}) \circ G_{k-1}(h_k(x), h_1, \ldots, h_{k-1})$$

The first few levels look like:

$$\begin{aligned} G_0(x) &= x \\ G_1(x, h_1) &= x\ h_1(x) \\ G_2(x, h_1, h_2) &= x\ h_1(x)\ h_2(x)\ h_1(h_2(x)) \end{aligned}$$

Thank you!