How to Construct Random Functions

Saujas Nandi

Based on: Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. J. ACM 33, 4 (August 1986).

Random Functions

- What do we want in a pseudorandom function generator?
  - Indexing: Picking a random function is easy
  - Polynomial-time Evaluation: Computation is easy
  - Pseudorandomness: Polynomial time algorithms cannot distinguish generated function from truly random function
- Prior Work
  - Focus was on random strings
    - Kolmogorov complexity: measure of randomness is length of shortest description
  - Already found method for generating random strings/sequences
    - Assuming one-way functions exist, there exists a polynomial-time algorithm that generates pseudorandom poly(k)-bit strings from k-bit inputs

Poly-Random Collection

- Set of functions that provides indexing, polynomial-time evaluation, and pseudorandomness
- Let I_k denote the set of all k-bit strings
- Why can't we index into the set of all functions from I_k to I_k?
  - The cardinality of the set is too big:
    - There are 2^k options to map each of the 2^k domain element to
    - Leads to $2^{k * 2^k}$ possible functions => Need an exponential number of bits to index
- Pick a 2^k sized subset of all I_k to I_k functions instead
  - Each function has a unique k-bit index
  - We still need to fulfill easy computation and pseudorandomness

Why do we need Pseudo-Random Collections?

- Potential alternatives: one-way functions, cryptographically strong pseudorandom bit (CSB) generators
- One way functions:
  - One-way functions have unpredictable, but not random inverses
  - RSA is believed to be a one-way scheme, yet having its inverses for *x* and *y* makes it easy to find its inverse at *xy*
  - Unwanted behavior from a “random” function
- CSB Generators
  - CSB Generators stretch a k-bit length input seed to a k^t-bit long pseudorandom output string
  - Problem with implementing a random oracle that maps a k^t sized subset of I^k to {0, 1}:
    - Need to store the result of each mapping so that oracle queries for the same string return the same result - Uses k^{t+1} bits of storage, k-bits for each of the k^t queries

CSB Generators

- Original definition
  - A polynomial time program that uses a random seed to generate a random string that passes all next-bit-tests: guessing the next bit in the sequence should be hard even if given prior bits
- Generalized definition:
  - For all probabilistic polynomial-time algorithm *T* that takes in *q* strings, each μ-bits long, and outputs 0 or 1, we know that for all sufficiently large k and any polynomial Q(k):

$$|p_k^s - p_k^r| < \frac{1}{Q(k)}$$

where p_k^s denotes the probability that *T* outputs 1 when strings are randomly found using CSB generators

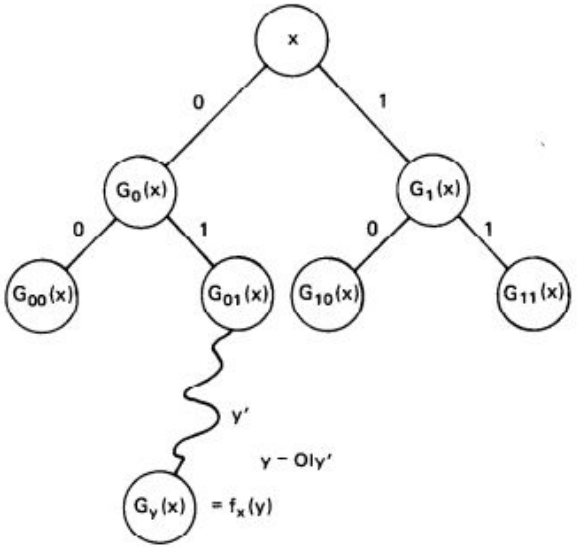
and p_k^r denotes the probability that *T* outputs 1 when strings are truly random

Construction

- Big Picture:
  - Assume the existence of one-way functions
  - Use any one-way function to construct a CSB generator (result given in a prior work by Levin)
  - Use this CSB generator to create a poly-random collection
- Intuition:
  - A CSB generator provides a way to generate good pseudorandom strings, so we can extend them to create pseudorandom functions
  - We will show that if an adversary can detect the usage of our pseudorandom functions, there is also an adversary that can detect the usage of a CSB generator

Construction

- Pick a CSB generator *G* that stretches a k-bit long seed into a 2k-bit long sequence
  $G(x) = b_1 \dots b_{2k}$
- Let $G_0(x) =$ first k-bits of G(x)
- Let $G_1(x) =$ last k-bits of G(x)
  For $\alpha = \alpha_1 \dots \alpha_t$
  let $G_\alpha(x) = G_{\alpha_t}(\dots G_{\alpha_2}(G_{\alpha_1}(x)) \dots)$
- For a function f_x indexed by *x*, we define $f_x(y) = G_y(x)$ where *y* is a k-bit long input
  - The poly-random collection is the set of all f_x's



From the figure, we can see that computing $f_x(y)$ will take a polynomial amount of time, since CSB generators run in polynomial time and we have to traverse down to the k-th depth

Pseudorandomness Proof

- Assume that there is a statistical test for functions *T* that a poly-random collection does not pass
  - Have $|p_k^F - p_k^H| > \frac{1}{Q(k)}$ for some k and polynomial *Q(k)*
  - Use *T* to construct a test for strings A_T s.t. the set of CSB sequences produced by *G* does not pass A_T - which is a contradiction

Pseudorandomness Proof

- Define oracle A_i as:

  if y is the first query with prefix $y_1 \cdots y_i$
  then A_i selects a string $r \in I_k$ at random, stores the pair $(y_1 \cdots y_i, r)$, and answers $G_{y_{i+1} \cdots y_k}(r)$
  else A_i retrieves the pair $(y_1 \cdots y_i, v)$ and answers $G_{y_{i+1} \cdots y_k}(v)$.

  where *y* is a k-bit long query string
  - Start with a full binary tree of depth *k* and store random k-bit strings in all level-*i* nodes, generate further levels using G_0 and G_1
  - Note that A_0 corresponds to using our poly-random collection as the oracle while A_k corresponds to using truly random functions as the oracle

Pseudorandomness Proof

- Let p^i be the probability that *T* outputs 1 when its queries are answered by A_i
  - Have $|p^0 - p^k| > 1/Q(k)$ for some k and polynomial Q(k)
- Construct A_T as follows:
  - For each 2k bits long query *y,* A_T first randomly picks a *i* uniformly between 0 and *k* - 1
  - Letting U_k denote the set of query strings, A_T answers T's oracle queries as follows:

if y is the first query with prefix $y_1 \cdots y_i$
then A_T picks the next string in U_k. Let $u = u_0u_1$ be such a string (u_0u_1 is the concatenation of u_0 and u_1, and $|u_0| = |u_1| = k$). Then A_T stores the pairs $(y_1 \cdots y_i0, u_0)$ and $(y_1 \cdots y_i1, u_1)$ and answers

$$G_{y_{i+2}\cdots y_k}(u_0) \quad \text{if} \quad y_{i+1} = 0 \quad \text{and} \quad G_{y_{i+2}\cdots y_k}(u_1) \quad \text{if} \quad y_{i+1} = 1.$$

else A_T retrieves the pair $(y_1 \cdots y_{i+1}, v)$ and answers $G_{y_{i+2}\cdots y_k}(v)$.

Pseudorandomness Proof

- If U_k is the set of CSB generated strings, A_T simulates the result of T with oracle A_i
- If U_k is the set of truly random strings, A_T simulates the result of T with oracle A_{i+1}
- The expected difference of probability that A_T outputs 1 when using CSB generated strings versus using truly random strings:

$$\sum_{i=0}^{k-1}(1/k)\cdot p^i - \sum_{i=0}^{k-1}(1/k)\cdot p^{i+1} = (1/k)(p^0 - p^k) > \frac{1}{kQ(k)}$$

Generalized Poly-Random Collections

- Sometimes we need to create random functions that map from $I_{p(k)}$ to $I_{q(k)}$ (rather than $I_{p(k)}$ to $I_{p(k)}$)
- Use two CSB generators:
  - G maps k-bit strings to 2k-bit strings
  - G` maps k-bit strings to q(k)-bit strings
- Instead of using $f_x(y) = G_y(x)$, use $f_x(y) = G'(G_y(x))$ where *y* is p(k)-bits long
- Similar proof

Polynomially-Inferable

- Algorithm can make a polynomial number of oracle calls before it must “infer” the result of some non-queried element
  - After P(x) oracle calls, algorithm A is given *f(x)* and a random bit, it must determine *f(x)* with at least 1/2 + 1/Q(k) probability where *P*, *Q* are polynomials and k is some sufficiently large value
- *F* cannot be polynomially inferred by any algorithm if and only if it passes all polynomial-time statistical tests for functions
  - If *F* can be polynomially inferred, easy to construct a polynomial-time statistical test that *F* cannot pass
  - Converse is harder to prove

Applications

- Protocol design:
  - Prove correctness assuming the existence of truly random functions
  - Replace truly random functions by functions randomly selected from poly-random collection
  - Maintains all properties of original protocol with respect to polynomially-bounded adversaries
- Used for message authentication, hashing, friend or foe identification, etc
  - Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1985. On the cryptographic applications of random functions. CRYPTO 84.