CS 880: Pseudorandomness and Derandomization 1/23/2013

Lecture 1: Introduction

Instructors: Holger Dell and Dieter van Melkebeek Scribe: Gautam Prakriya

This lecture provides a brief discussion of the power of randomness in computing, presents the main topics of this course, and introduces the central notion of pseudorandom distributions.

1 The Power of Randomness in Computing

In theoretical computer science randomness refers to the ability to toss a fair coin, or equivalently, to draw a uniform sample from $\{0,1\}^r$, where r denotes the number of coin flips needed. In practice we do no have access to such a perfect source of unbiased and uncorrelated random bits, but let us assume for now that we do and investigate the power it gives us. In some computational settings random bits turn out to be essential – either for fundamental reasons or for reasons of efficiency – but in some other settings they are conjectured to be dispensable.

1.1 Essential for fundamental reasons

In some settings randomness is inherent in the task at hand and/or there provably are no deterministic solutions. Examples include the following.

- Key generation in cryptography.
  Generating a key deterministically is insecure because anyone else could retrieve and use the same key.

- Symmetry breaking in concurrent programming.
  Symmetry breaking is often a key bottleneck for problems in concurrent programming, and in a number of situations randomness is provably needed. One example is the leader election problem, in which a network of processes aim to designate a single process as the leader. If the underlying network graph contains certain symmetries deterministic leader election is impossible [Ang80], but there are simple randomized approaches that work for all graphs.

1.2 Essential for efficiency reasons

In some settings there exist deterministic solutions but they are provably more expensive than the best randomized solutions. Examples include the following.

- Approximating the average of a function given as a black-box.
  Suppose you are given a black-box for a function $f : \{0,1\}^n \mapsto \{0,1\}$, and want to compute an approximation to the average, $\frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)$, that is no more than .01 off.

  A deterministic algorithm will need to query f in $\Omega(2^n)$ points because otherwise the value of f can be changed at the unqueried points so as to make the output more than .01 off. On the other hand, as we will analyze later in the course, taking the average of f at a sufficiently large constant number of uniform samples from $\{0,1\}^n$ yields an accurate answer with very high probability.

- Communication in distributed computing.
  Suppose Alice holds an input $x \in \{0,1\}^n$, Bob holds an input $y \in \{0,1\}^n$, and Alice wants to send a short message to Bob so that Bob can figure out whether $x = y$.

  Using any deterministic strategy, Alice will need to send n bits in the worst case. If she sends fewer, there are two inputs for Alice, say x_1 and x_2, on which Alice sends the same message to Bob. In that case, Bob will come to the same conclusion for input pairs $(x, y) = (x_1, x_1)$ and $(x, y) = (x_2, x_1)$, whereas the correct answers for those input pairs are opposite.

  On the other hand, there is a randomized strategy in which Alice only sends $O(\log n)$ bits to Bob and Bob arrives at the correct conclusion with 99% certainty. The idea is for Alice to hash her input x, send the hashed value as well as the specification of the hash function to Bob, and then Bob checks whether his input hashes to the same value. More specifically, for some sufficiently large constant c, Alice picks a random prime p of $c \cdot \log n$ bits, interprets her input x as a number between 1 and 2^n, computes $h = x \bmod p$, and sends h and p to Bob. Bob checks whether $h = y \bmod p$; if so, Bob answers "yes"; otherwise, Bob answers "no".

  When $x = y$, Bob always correctly answers "yes". When $x \neq y$, Bob answers "no" unless p happens to be one of the prime divisors of $x - y$. Since there can be no more than n such prime divisors and there are more than $100n$ primes of $c \log n$ bits when the constant c is sufficiently large, the odds that Bob gives an incorrect answer are less than 1%.

Both of the above examples have an information-theoretic flavor. As we'll see next, this is not a coincidence.

1.3 Provably or conjectured dispensable

In purely computational settings, randomness can often be eliminated at a small cost. In some cases this is provably so, i.e., we can construct efficient deterministic simulations that do not blow up the required resources such as time or work space by much. Recent success stories in this context include the following.

- Primality testing.
  Testing whether a number is prime plays an important role in many cryptographic protocols. The trivial trial-division algorithm and its improvements like the Sieve of Eratosthenes take time exponential in the bit length of the number. In the 1970's several randomized polynomial-time algorithms were devised [SS77, Mil76, Rab80], but deterministic polynomial-time algorithms remained elusive for a long time. A few years ago such an algorithm was finally developed [AKS04].

- Connectivity in undirected graphs.
  Given an undirected graph on n vertices and two vertices s and t, does there exist a path connecting s and t? Breath-first search and depth-first search solve this problem in linear time but require $\Omega(n)$ bits of work space in order to keep track of which vertices have already been visited. On the other hand, consider a random walk starting from s. If s and t are not connected, the random walk will never hit t. If s and t are connected, one can show that t will be hit with very high probability within n^3 steps [AKL$^+$79]. This yields a randomized algorithm that takes longer than breadth-first or depth-first search but only needs $O(\log n)$ bits of work space, namely to store the current vertex and count up to n^3. Deterministic

algorithms that only need $O(\log n)$ bits of work space were open for a quarter of a century but recently discovered [Rei08] – and we will cover one in this course.

We should point out that in both examples the randomized algorithms are arguably simpler and more elegant than the deterministic ones, and are still somewhat more efficient – by a polynomial time factor in the case of primality, and a constant space factor in the case of undirected connectivity. However, from a complexity-theoretic perspective these advantages are not significant.

For several other problems we have strong evidence but as of yet no proofs that an existing efficient randomized algorithm can be derandomized at small cost. After primality testing was resolved, the following problem established itself as the new holy grail in derandomization.

- Polynomial identity testing.
  There are several variants of this problem but the most natural one is the following: Given an arithmetic formula consisting of integer variables, additions, subtractions, and multiplications, is the formula identically zero? For example,

  $$(x_1 + x_2) \cdot (x_1 - x_2) - x_1 \cdot x_1 + x_2 \cdot x_2$$

  is a positive instance, and x_1 is a negative one.

  There is a very simple randomized algorithm – plug in random values for each of the variables, and see whether the formula evaluates to zero on those values. If it does not, the answer is definitely negative. If it does, chances are the answer is positive because a formula that is not identically zero can only evaluate to zero on a small fraction of the points. This follows from a multivariate generalization of the fact that a non-zero univariate polynomial of degree d can have no more than d roots.

  Lemma 1 (Schwartz–Zippel). *Let f be a multivariate polynomial over the integers*[1] *with n variables and of total degree d that is not identically zero, and let R be a non-emtpy set of integers. Then* $\Pr_{x \in_u R^n}[f(x) = 0] \leq \frac{d}{|R|}$.

  Exercise 1. *Prove the Schwartz–Zippel Lemma by induction on n.*

  The set R in the lemma denotes the range from which the random values are chosen, and the notation $x \in_u R^n$ means that x is chosen uniformly at random from R^n. Since the degree of the polynomial computed by a given arithmetic formula is bounded by the length of the formula, the range R need not be large in order to keep the probability of error small, and evaluating the formula on x does not involve large values.

In spite of the simplicity of the resulting randomized algorithm for polynomial identity testing, no polynomial-time deterministic algorithm is known to date. However, the conjecture is that such algorithm exists. In fact, the common belief is that *every* randomized decision procedure (i.e., a randomized algorithm for a yes/no problem) can be simulated deterministically with only a polynomial overhead in running time, and only a constant-factor overhead in work space. We will briefly discuss the evidence for this belief in the next lecture and study it in more detail later in the course.

[1] The lemma actually holds more generally for polynomials over any integral ring.

2 Main Topics

The two main topics in this course are derandomization and randomness extraction.

Derandomization. Like time and work space, random bits can be viewed as a computational resource. Given the practical difficulty in generating purely random strings, it is arguably an expensive resource. A major goal in complexity theory is to reduce the dependency on (pure) randomness where possible. We will study various approaches to do so, and investigate the extent to which they are successful. In some settings it is possible to achieve full derandomization, i.e., efficient deterministic simulations of the randomized process. In some other setttings only partial derandomizations are known.

Randomness extraction. When designing and analyzing randomized algorithms, we typically assume a "pure" random source, i.e., one that produces a uniform sample from $\{0,1\}^r$. Available sources of randomness (if any) are not pure. Examples include clock times, communication delays over the internet, and noise caused by avalanche breakdown in Zener diodes. These are rather crude sources of randomness in that the bits they produce often exhibit significant biases and correlations. In order to run our randomized algorithms using such a source, we first need to transform the crude source into a pure source or an almost-pure source. This step is called randomness extraction. We will see how to do it for a very broad class of crude sources.

3 Pseudorandom Distributions

The standard way of derandomization involves the construction of distributions that can be generated using little randomness but nevertheless look like a truly random distribution to the process under consideration. Such distributions are called pseudorandom. We need to review some probability notation before making the notion precise, but let us stress from the outset that the notion of pseudorandomness is always relative to a class $\mathcal{A}$ of randomized algorithms.

Probability notation. We consider distributions over the set Σ^* of strings over some alphabet Σ, typically the binary alphabet $\Sigma = \{0,1\}$. We write $\rho \leftarrow D$ to denote that ρ is chosen according to the distribution D. When we don't need a name ρ for the sample, we sometimes simply write D to denote a sample from D. For example, for any set T we write $\Pr[D \in T]$ to denote $\Pr_{\rho \leftarrow D}[\rho \in T]$. For $r \in \mathbb{N}$, we use U_r to denote the uniform distribution on $\{0,1\}^r$, i.e., for all $z \in \{0,1\}^r$ we have $\Pr[U_r = z] = 1/2^r$.

We use the statistical distance $d_{\text{stat}}(D, D')$ to measure how far two distributions D and D' over Σ^* are from each other. It equals the largest difference in probability D and D' assign to an event $T \subseteq \Sigma^*$:

$$d_{\text{stat}}(D, D') = \max_{T \subseteq \Sigma^*} \Big| \Pr[D \in T] - \Pr[D' \in T] \Big| .$$

This maximum is achieved when T consists of all $y \in \Sigma^*$ that have a higher probability under D than under D', and shows that

$$d_{\text{stat}}(D, D') = \frac{1}{2} \sum_{y} \Big| \Pr[D = y] - \Pr[D' = y] \Big| .$$

In other words, the statistical distance also equals half the 1-norm of the difference of the two distributions when we view them as vectors of probabilities: $d_{\text{stat}}(D, D') = \frac{1}{2}||D - D'||_1$. Its value is always in the range $[0, 1]$, where the lower bound holds if and only if the distributions are identical, and the upper bound holds if and only if the distributions have disjoint supports.

Pseudorandomness. We are now ready to formally define the critical concept of a *pseudorandom distribution* for a class $\mathcal{A}$ of algorithms A. In this context by a distribution D we really mean a sequence of distributions $D = (D_r)_{r\in\mathbb{N}}$ where D_r is a distribution on $\{0,1\}^r$. We use n to indicate the input length to A, and r the number of random bits A needs on input x. For $x \in \Sigma^n$ and $\rho \in \{0,1\}^r$, we denote by $A(x, \rho)$ the output A produces on input x when ρ is used as the random bit sequence, and write $A(x, D_r)$ to denote the distribution of $A(x, \rho)$ when ρ is sampled from the distribution D_r.

Apart from the underlying class $\mathcal{A}$ of algorithms, there is an additional parameter ϵ, which bounds how much the distribution D can "look different" from the uniform distribution $U \doteq (U_r)_{r\in\mathbb{N}}$ to algorithms from $\mathcal{A}$. We allow ϵ to be a function of r.

Definition 1 (Pseudorandom distribution). *D is $\epsilon(r)$-pseudorandom for $\mathcal{A}$ if* $(\forall A \in \mathcal{A})(\forall^\infty x \in \Sigma^*)$

$$d_{stat}(A(x, U_r), A(x, D_r)) \leq \epsilon(r). \tag{1}$$

In words: For every algorithm A from $\mathcal{A}$ and for all but finitely many[2] inputs x, using the pseudorandom distribution instead of the uniform distribution as the random bit source, results in an output distribution that has statistical distance no more than ϵ from the true output distribution of A on input x. In the case of decision procedures A, condition (1) can be written equivalently as

$$\Big|\Pr[A(x, U_r) \text{ accepts}] - \Pr[A(x, D_r) \text{ accepts}]\Big| \leq \epsilon\,.$$

What we aim for are pseudorandom distributions D such that D_r can be "generated" from $\ell(r) \ll r$ truly random bits in that there exists a transformation $G_r : \{0,1\}^{\ell(r)} \to \{0,1\}^r$ such that $D_r = G_r(U_{\ell(r)})$. We will refer to such a transformation as a pseudorandom generator (PRG) with short seed length. Its existence implies that the support of D_r is only a tiny fraction of the support of U_r, and that $d_{\text{stat}}(U_r, D_r)$ is close to its maximum value of one. Nevertheless, the distributions $A(x, U_r)$ and $A(x, D_r)$ are very close in statistical distance. This is because the algorithm A only uses its randomness in a certain way.

Choices of $\mathcal{A}$. For an individual algorithm A such as one of the known randomized primality tests, one can try to analyze what properties are needed of the random source, and exploit those to obtain good pseudorandom distributions for $\mathcal{A} \doteq \{A\}$. Rather than singleton classes $\mathcal{A}$, in this course we are more interested in large classes of algorithms, so as to obtain generic derandomizations that are oblivious to the details of the algorithm.

In cryptographic settings one often picks $\mathcal{A}$ to be the class of all polynomial-time algorithms. This is because all involved parties are assumed to be polynomial-time agents but we have no knowledge of the degree of the polynomials. In the setting of time-bounded derandomization, we

[2]Requiring the condition for *all* inputs x turns out to be too restrictive for many choices of $\mathcal{A}$, and "all but finitely many" is equally good from the asymptotic perspective of complexity theory.

typically do know the degree d of a polynomial that bounds the running time, and therefore consider the class of all decision algorithms that run in time n^d. Similarly, in the setting of space-bounded derandomization, we typically know a constant d' such that the algorithm only needs $d' \cdot \log n$ bits of work space, and we consider the class of all such decision algorithms.

At first it may seem daunting to construct pseudorandom generators with short seed length for such large resource-bounded classes. One reason why there is hope is the following mantra:

Whether something looks random to you, depends on your computational power.

For example, if someone shows you a sequence of numbers and asks you what the next number in the sequence is, at first you may have no clue at all – the sequence looks completely random to you. But the sequence can be the result of a complex deterministic process, and when you are given more time or other resources (like a smart friend), you may be able to discern a pattern and predict what the next number is. Similarly, a distribution that is generated from few truly random bits and therefore very far from uniform in statistical distance, may look like a random distribution to algorithms that have few resources but not to algorithms with more resources.

Connections between hardness and randomness as in the above mantra form an important theme in this course. We will expand on them a bit more in the next lecture, and develop them in detail later on, for the above classes $\mathcal{A}$ as well as for some more restricted but still quite broad classes.

To end this lecture, we observe that if we pick $\mathcal{A}$ as the class of all algorithms, then the condition of D being $\epsilon(r)$-pseudorandom is equivalent to D_r having statistical distance at most $\epsilon(r)$ from uniform at all but finitely many lengths r. Such distributions cannot be generated from $\ell(r) \ll r$ truly random bits. In other words, there are no PRGs with short seed length for the class $\mathcal{A}$ of all algorithms. In particular, the PRGs that are used to generate randomness in higher-level programming languages all have their limitations. They may work fine most of the time in practice, but there are documented cases where the results obtained are horribly wrong due to the limitations of the PRG [FLW92, GW96]. Whereas in higher-level programming languages the limitations of the PRGs are usually shoved under the rug, in this course we will always carefully specify the class $\mathcal{A}$ for which the generated distribution is supposed to be pseudorandom, and rigorously prove that this is the case before using it.

References

[AKL+79] R. Aleliunas, R. Karp, R. Lipton, L. Lovsz, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science, FOCS 1979*, pages 218–223, 1979.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[Ang80] D. Angluin. Global and local properties in networks of processors. In *Proceedings of the 12th Annual Symposium on Theory of Computing, STOC 1980*, pages 82–93, 1980.

[FLW92] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384, 1992.

[GW96] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr. Dobbs Journal*, pages 66–70, 1996.

[Mil76] G. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[Rab80] M. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[Rei08] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4), 2008.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.