Randomness, Pseudorandomness and Derandomization Sept 14th, 2010

Lecture Notes 1: Randomness - A computational complexity view

Professor: Avi Wigderson (Institute for Advanced Study) *Scribe: Dai Tri Man Lê*

1 Computational Complexity

1.1 Hard vs. easy problems

The problem of multiplying two integers is clearly an easy problem since the simple grade school multiplication algorithm can return the product of two n digit numbers in $O(n^2)$ steps. A problem is *easy* if it can be solved by a polynomial time algorithm, i.e., algorithm that runs in polynomial steps with respect to the size of the input. The class of all problems that can be solved using a polynomial time algorithm is denoted by P.

Next consider the FACTORING problem, where we want to split an integer into two smaller non-trivial divisors, which when multiplied together equal the original integer. The best known algorithm for factoring takes $O(exp(\sqrt{n}))$ steps on an n digit input. It is open whether factoring has a polynomial time algorithm or not. So we don't know if factoring is a hard problem. However, it is widely believe that factoring is hard since the theory of public key cryptography depends on the hardness of factoring. We call a problem *hard* if there is no polynomial time algorithm solving it.

1.2 The P vs. NP question

Consider the MAP COLORING problem, where we take input as a planar map M with n countries. We observe the following questions:

- 2-COLORING: Is M 2-colorable?
- 3-COLORING: Is M 3-colorable?
- 4-COLORING: Is M 4-colorable?

The 2-COLORING problem is clearly easy since we answer using a simple greedy algorithm. The 4-COLORING problem is extremely easy since the answer is always yes for every planar graph by the *four color theorem*. However, it remains unknown if the 3-COLORING problem is easy or not, but we know the following theorem:

Theorem 1 (Cook-Levin '71, Karp '72). 3-COLORING *is* NP*-complete.* ♦

Here NP stands for *nondeterministic polynomial time*. Intuitively, NP is the class of problems whose solutions can be verified easily. For example, for 3-COLORING, once given a coloring of the map M, we can easily check in polynomial time if that coloring is valid by checking if any two adjacent countries have different color.

A problem in NP is called NP-complete if and only if we can reduce any other NP problems to it by a polynomial time transformation of the inputs. In fact, many problems in all sciences and engineering are NP-complete.

Note that we know FACTORING is in NP since we can easily check if the product of two integers is equal to another integer. But it remains open whether or not FACTORING is NP-complete.

The most fundamental question, i.e., the P vs. NP question, in computational complexity can be stated as following: is the 3-COLORING problem (or any other NP-complete problem) easy? Or informally it is the question: can *creativity* be automated? More generally we want to know how fast can we solve:

- The FACTORING problem
- Map coloring
- Satisfiability of Boolean formulae
- Computing the Permanent of a matrix
- Computing optimal Chess/Go strategies
- ...

The best known algorithms for these problems seem to take exponential time/size. But is exponential time/size necessary for some? This is the most fundamental question of computational complexity. Most of complexity theorists believe that all NP-complete problems are hard, i.e., $\mathsf{P} \neq \mathsf{NP}$. Actually complexity theorists believe the following stronger conjecture:

Conjecture 1. All NP-complete problems require exponential size circuits to compute. ♦

2 The power of randomness in saving time

We will see a host of problems, which we have *probabilistic polytime* algorithm, but (still) have no deterministic polytime algorithms.

The main idea is to introduce randomness into the polytime algorithm, and we only require good probabilistic algorithm to succeed with high probability, e.g., with 99.99% probability. But one might ask why we tolerate errors? The reasons are:

- we tolerate errors in life
- we can reduce the probability of errors to arbitrarily small ($< exp(-n)$) by repetition.
- to compensate, we can do much more...

We next have a look at well-known probabilistic algorithms.

NUMBER THEORY: PRIMES. Consider the following problem asked by Gauss: given $n \in [2^n, 2^{n+1}]$ is x prime? Two simple and fast probabilistic algorithms were invented in 1975 by Solovay-Strassen and Rabin. A later breakthrough is the discovery of polynomial time deterministic algorithm, aka AKS algorithm, for primality testing in 2002 due to Agrawal, Kayal and Saxena, but the AKS algorithm is not as efficient, and thus not often used in practice.

However, no deterministic polynomial time algorithm is known for the following closely related problem: given n, find a prime in $[2^n, 2^{n+1}]$. But we can solve the problem using the following simple probabilistic algorithm: pick at random a sequence of random numbers $x_1, x_2, \ldots, x_{100n}$ from $[2^n, 2^{n+1}]$, and for each x_i apply primality test. By the *prime number theorem*, we can easily show that the $\mathsf{Prob}[\exists i, x_i \text{ is a prime}]$ is very high.

ALGEBRA: POLYNOMIAL IDENTITIES. For example, we want to check if

$$det\left(\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{bmatrix}\right) - \prod_{1 \le i < j \le n} (x_j - x_i) \equiv 0?$$

We know from a theorem by Vandermonde, the answer is yes. But assume that we don't know this theorem, how do we check if this identity is true?

In general, given (implicitly, e.g. as a formula) a polynomial $p(x_1, \dots, x_n)$ of degree d, we want to known if $p(x_1, \dots, x_n) \equiv 0$. The following probabilistic algorithm by Schwartz-Zippel from '80 solves the problem: pick r_i independently at random from $1, 2, \dots, 100d$. Then

$$p \equiv 0 \Rightarrow \mathsf{Prob}[p(r_1, \dots, r_n) = 0] = 1$$
$$p \not\equiv 0 \Rightarrow \mathsf{Prob}[p(r_1, \dots, r_n) \neq 0] > .99$$

Again no polynomial time deterministic algorithm is known for this problem. Polynomial identity testing has applications in program testing.

ANALYSIS: FOURIER COEFFICIENTS. Given (implicitely) a function $f : (Z_2)^n \to \{-1, 1\}$ (e.g. as a formula), and $\epsilon > 0$, find all characters χ such that $|\langle f, \chi \rangle| > \epsilon$. Observe that there are at most $1/\epsilon^2$ such χ. Thus using adaptive sampling technique Goldreich and Levin in '89 introduced a probabilistic algorithm solving this problem, which succeeds with high probability. This method was later generalized to other Abelian groups.

These algorithms have applications in coding theory, complexity theory, learning theory, game theory. A polytime deterministic algorithm is also not known for this problem.

GEOMETRY: ESTIMATING VOLUMES. Given (implicitly) a convex body K in $\mathbb{R}^d$, where d is large, e.g. by a set of linear inequalities. Estimate *volume*(K). Computing *volume*(K) exactly is very hard since it is $\#\mathsf{P}$-complete. $\#\mathsf{P}$ is defined to be the class of counting functions $f : \sigma^* \to \mathbb{Z}$, such that there exists a polynomial time bounded Turing machine M such that for all x, $f(x) = |\{y \mid M(x, y) \text{ accepts}\}|$.

However, a probabilistic algorithm for this algorithm was given by Dyer-Frieze-Kannan in '91. To simplify our explanation, consider the case of $\mathbb{R}^2$. The idea is to put the body K inside a grid, and then use the random sampling technique to approximately count the number of cells inside the body K, which gives the approximate volume of K. Randomly sampling was done by taking a random walk inside K. The correctness proof of this algorithm uses rapid mixing of Markov chains and the connection between spectral gap and isoperimetric inequality. This algorithm has found applications in areas like statistical mechanics and group theory.

This leads us to the second most fundamental question of computational complexity. We want to know whether or not randomness helps in saving time. In other words, are there problems with probabilistic polynomial time algorithm but no deterministic one? One might conjecture that:

Conjecture 2. There exists a problem that can be solved with a probabilistic polynomial time algorithm but not with a deterministic polynomial time algorithm. Formally, $\mathsf{BPP} \neq \mathsf{P}$. ♦

Surprisingly, a theorem by Impagliazzo-Wigderson in '97 together with an observation by V. Kabanets suggest that Conjecture1 and Conjecture 2 can not be both true! This point can be explained more clearly when we discuss the so called "hardness vs. randomness" paradigm in the next section.

3 The hardness vs. randomness paradigm and the weakness of randomness

This "hardness vs. randomness" paradigm is the result of almost 20 years of development by many researchers (Blum-Micali, Yao, Nisan-Wigderson, Impagliazzo-Wigderson et al.) starting with questions in cryptography. The basic and informal intuition is that if there are natural hard problems ("natural" in the sense that problems are in NP or $\#\mathsf{P}$), then we can use this hardness to generate sequences that look random in a deterministic fashion, and thus randomness can be efficiently eliminated from probabilistic algorithms.

More formally, we will state one of the strongest results we have in this "hardness vs. randomness" paradigm:

Theorem 2 (Impagliazzo-Wigderson '98). *If there exists a problem, say* $\mathsf{SAT} \in \mathsf{NP}$*, in exponential time that requires exponential size circuits, then every probabilistic polynomial-time algorithm has a deterministic counterpart.*

The Impagliazzo-Wigderson Theorem essentially gives us one directional connection: "hardness implies randomness". In fact, there has been some work showing a partial converse of Theorem 2:

Theorem 3 (IKW '04, Impagliazzo-Kabanets '05). *Derandomization implies hardness.*

We will not discuss more about this converse direction in this talk. But we will try to give the main idea of how we can show the "hardness implies randomness" direction. Note that the Impagliazzo-Wigderson Theorem seems extremely strong since it claims that it can deramdomize all probabilistic polynomial-time algorithms, including the ones that are not yet known and invented. How are we supposed to prove that?

The key idea lies in the notion of "computational pseudorandomness". Observe that a generic probabilistic polytime algorithm can be seen as a polytime algorithm $A(x, r)$ taking two inputs. The first input x is the input of the function you want to compute, e.g., the number of you want to test for primality etc. And the second input r is the sequence of random (unbiased and independent) bits. Our goal is to run this algorithm without the truly random input! To do so, we need to understand how the probability of the output of the algorithm changes if the random sequence input is not drawn from a uniform distribution (i.e., the bits might be biased and dependent). Thus, the absolute key is to understand which distribution can fool this probabilistic algorithm and cause it to behave as if the random input is taken from a perfect uniform distribution. We will call these distributions *pseudorandom*.

Definition 1 (Goldwasser-Micali '81). *A distribution D is* pseudorandom *if for every "efficient" algorithm $A(x, r)$, for every input x, the difference between the probability that $A(x, r)$ outputs 1 with r sampled from D and the probability that $A(x, r)$ outputs 1 with r sampled from a truly uniform distribution is negligible.* ♦

In other words, no efficient algorithm can distinguish a pseudorandom distribution from a uniform distribution! Thus, it suffices to be able to generate these pseudorandom distributions. However, we do not know if such pseudorandom distribution with such low, says zero, entropy can be generated deterministically and efficiently. So we relax our goal by assuming that we have very few (logarithmic) truly random bits (seed), we want to have a deterministic efficient construction that generates much longer (polynomial length) pseudorandom bits. Such a construction is called a *pseudorandom generator*.

Note that using a pseudorandom generators, we can derandomize a probabilistic polytime algorithm trying to run it on all possible outputs of a pseudorandom generator for all possible logarithmic-length random seeds, and then take the majority vote. Since all the seeds have logarithmic length, this derandomized version of the algorithm still takes polynomial time.

Thus, it remains to show how we can build such a pseudorandom generator, and here is where we need the hardness assumption. For simplicity of this talk, we will only discuss the construction of a "toy" pseudorandom generator that can extend a truly random seed of length k to produce a pseudorandom string of length $k+1$, assuming that we have a hard function $f : \{0,1\}^k \to \{0,1\}$. In this case, given a random seed $s \in \{0,1\}^k$, it is not hard to verify that the sequence consisting of s followed by the bit produced by $f(s)$ is pseudorandom. Intuitively, if there is an efficient algorithm that can distinguish this pseudorandom last bit $f(s)$ with a truly random bit, then it can also computes the hard function, which is a contradiction.

It is interesting to note that although we haven't been able to prove that a hard function exists, in many cases, we can derandomize a specific probabilistic polytime algorithm without the hardness assumption. One famous example is the AKS algorithm for primality testing mentioned previously. The solution of this primality testing problem really fits into the framework discussed here. First, a new randomized algorithm was designed by Agrawal and Biswas in 1999 for primality testing. Then, Agrawal, Kayal and Saxena was able to analyze and understand the way the algorithm used the random bits well enough to design a "pseudorandom generator" specifically made to derandomize this algorithm.

4 The power of randomness in other settings

In the rest of this talk, we will turn to some other advantages of randomness in algorithms, beyond that of saving time. We will also see some situations where randomness are essential.

4.1 Randomness and space complexity

A famous example randomness can be used to save space is the reachability problem on undirected graphs. Imagine we are lost in a city without a map, and we want to get back to our hotel! It turns out that there is a simple randomized logarithmic-space algorithm due to Aleliunas, Karp, Lipton, Lovász, and Rackoff in '79 that by simply taking a random walk will lead us back to our hotel in polynomial time. Actually, in polytime we will be able to visit every hotel in the city, and not just our hotel using a random walk. A deterministic log-space algorithm for undirected reachability was an open problem for nearly 25 years, and was resolved recently by Reingold in 2005, where the source of pseudorandomness is from a very different source called *expander graphs* which we consider in the third lecture.

4.2 The power of randomness in Proof Systems

Two other famous, and striking, applications of randomness lie in interactive proof systems, where a verifier is trying to check a claim that a prover can prove a given statement S (e.g. the Riemann hypothesis). More formally, the verifier employs some algorithm V that takes S as input, performs some queries to the prover, and then returns a boolean output. A deterministic verifier should satisfy the following two properties:

1. Soundness: if the prover does not have a proof of S, then V rejects.
2. Completeness: if the prover does indeed have a proof of S, then V accepts.

A probabilistic verifier is defined similarly, except that if the prover does not have a proof of S, we only require V to reject with very high probability.

Constructing a verifier with the soundness and completeness properties is trivial since the prover can simply hand over the entire proof to the verifier and the verifier can then check all the steps of the proof. Thus, if the proof has length n, then this procedure will take time $O(n)$. However, Arora, Lund, Motwani, Safra, Sudan and Szegedy in '90 showed a remarkable fact that a probabilistic verifier only needs to see a very tiny portion of the proof to verify it!

Theorem 4 (PCP theorem). *Every proof of a statement S of length n in a formal system can be converted in polytime into another proof in a special format which can be verified by a probabilistic verifier in $O(\log n)$ time using only constant number of queries to the prover.*

The PCP theorem has many applications in computational complexity, especially in the theory of *computational hardness of approximation*.

In a rather different direction, instead of verifying a proof as quickly as possible, one can consider the prover's desire of wanting to have his or her claim of proving S verified without allowing the verifier to learn enough of the proof to publish it first. Another remarkable fact is that this is possible (under some plausible complexity assumption). It was shown in [Goldwasser-Micali-Rackoff '85] and [Goldreich-Micali-Wigderson '86] that assuming factoring is hard, there exists a polynomial time probabilistic verifier for proofs of a statement S of length n which is *zero-knowledge* in the sense that the verification algorithm does not tell the verifier anything that the verifier did not already know, other than that the statement S is provable. We will discuss more about zero-knowledge proofs in the second talk.

(End of the first talk.)