CS265/CME309: Randomized Algorithms and Probabilistic Analysis

Lecture #1:Computational Models, and the Schwartz-Zippel Randomized Polynomial Identity Test

Gregory Valiant*, updated by Mary Wootters

January 1, 2025

1 Introduction

Welcome to CS265/CME309!! This course will revolve around two intertwined themes:

- Analyzing random structures, including a variety of models of natural randomized processes, including random walks, and random graphs/networks.
- Designing random structures and algorithms that solve problems we care about.

By the end of this course, you should be able to think precisely about randomness, and also appreciate that it can be a powerful tool. As we will see, there are a number of basic problems for which extremely simple randomized algorithms outperform (both in theory and in practice) the best deterministic algorithms that we currently know.

2 Computational Model

During this course, we will discuss algorithms at a high level of abstraction. Nonetheless, it's helpful to begin with a (somewhat) formal model of randomized computation just to make sure we're all on the same page.

Definition 2.1 A *randomized algorithm* is an algorithm that can be computed by a Turing machine (or random access machine), which has access to an infinite string of uniformly random bits. Equivalently, it is an algorithm that can be performed by a Turing machine that has a special instruction "flip-a-coin", which returns the outcome of an independent flip of a fair coin.

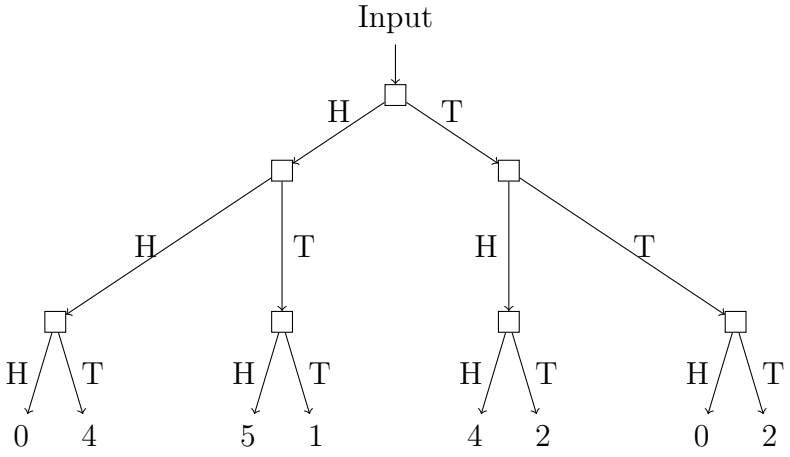
We will never describe algorithms at the level of Turing machine instructions, though if you are ever uncertain whether or not an algorithm you have in mind is "allowed", you can return to this definition. The above definition implies the following two important properties of randomized algorithms:

- Because the computation can depend on the randomness, the output of the algorithm, and the computation path itself, are random variables. As such, it makes sense to ask questions like "What is the probability the output is correct?" or "What is the standard deviation of the runtime of the algorithm." For these questions, we are thinking of the computation path as being a random variable, defined in terms of the randomness of the random bits that the algorithm uses.

  Of course, the output of the algorithm depends on both the randomness of the algorithm and also the input. We will typically view the input as fixed in advance. For example, if we are running a randomized primality testing algorithm to test if the input $q = 1173$ is prime, we will regard q as a fixed parameter, and analyze the execution of the algorithm on input $q = 1173$ in terms of the random coin flips. Frequently, we will be able to prove guarantees that hold for *all* inputs; namely, continuing the primality testing example, we will be able to say things like "for every input, q, with probability at least $1 - 1/2^{100}$ over the random coin flips of the algorithm, the algorithm will correctly output whether or not q is prime."

- Because we are working with a model where the algorithm has access to coin-flips (which take one of two possible outcomes), the computation path can be described as a *binary* tree, where the branches correspond to the outcomes of the coin tosses. The topmost branch corresponds to whether the first coin flip landed heads or tails, the next level corresponds to the second coin-flip, etc.

  For example, the tree might look like this, if the algorithm flips three coins and then outputs a number in $\{0, 1, 2, 3, 4, 5\}$.



Notice that if we have a randomized algorithm that terminates after flipping at most k coins, the probability of every output must be an integer multiple of $1/2^k$. (If this isn't obvious to you, take 3 minutes and convince yourself of it now...) This might be concerning if you'd like to consider a randomized algorithm that, say, executes command X with probability $1/3$. It turns out that this is fine: one can simulate a coin that lands heads with probability $1/3$ using a sequence of flips of a fair coin. However, if you are ever designing a randomized algorithm that uses funny transition probabilities, you can always come back to this model, and double-check whether what you are doing is actually feasible in this model.

There are two broad classes of randomized algorithms, referred to as "Las Vegas" algorithms and "Monte Carlo" algorithms:

- Las Vegas Algorithms: These algorithms always output the correct answer, though the runtime can be unbounded, though we require that it has finite expectation. (I.e. the output is deterministic, but the runtime is a random variable, with finite expectation.)
- Monte Carlo Algorithms: These algorithms have bounded runtime, though they can output the wrong answer. For algorithms that answer yes/no questions (e.g. "Is q prime?"), we group these into two subclasses, according to whether they have 1-sided or 2-sided error:
    1. 1-sided error: If the true answer is "Yes" then the algorithm will output "Yes", and if the true answer is "No", the algorithm will output "No" with probability at least $p > 0$. [We could also swap the Yes/No and it would still be called 1-sided error...] For the complexity theorists out there, the class of problems that have polynomial-time 1-sided error Monte Carlo algorithms is known as "RP" (Randomized-Polynomial).
    2. 2-sided error: The algorithm is correct with probability at least $1/2 + \epsilon$ for some $\epsilon > 0$. The class of problems that have polynomial-time 2-sided error Monte Carlo algorithms is known as "BPP" (Bounded Error Probabilistic Polynomial).

The definitions above guarantee only a slight bias towards being correct. Is this really good enough? One reason why this is enough is that we can increase the probability of correctness by running the algorithm multiple times. For example, given a 1-sided error Monte Carlo algorithm, if we run it t times, and it ever outputs "No" we can safely output "No", and if it always outputs "Yes", then we can output "Yes" and the probability we are incorrect is at most $(1-p)^t$. [Make sure you understand why this is true!!] We can bound

$$(1-p)^t \leq e^{-pt},$$

using the fact that $1-x \leq e^{-x}$ for all x so if we choose t to be, say, $10/p$, then the probability that we are incorrect is at most e^{-10}, which is a very small number.

This trick of replacing $1-x$ with e^{-x} is a commonly used trick for getting rid of a tricky minus sign, and replacing it with a nice exponent. For small values of x, this approximation is quite accurate, because the derivative of $1-x$ and e^{-x} are equal at $x = 0$.

For the 2-sided error algorithm, intuitively, the right approach is to repeat the algorithm t times, and output the majority answer. We're probably going to skip the following calculation in class, but it is included here for reference—we expect you to be able to do these sorts of calculations on your own.

Lemma 2.2 *If an algorithm is correct with probability $1/2 + \epsilon$, and we run it t times and output the majority answer, the probability we are correct is at least $1 - e^{-2t\epsilon^2}$.*

Proof:

$$\begin{aligned} Pr[\text{majority is wrong}] &\leq \sum_{i=0}^{t/2} \binom{t}{i} (\frac{1}{2}+\epsilon)^i (\frac{1}{2}-\epsilon)^{t-i} \\ &\leq \sum_{i=0}^{t/2} \binom{t}{i} (\frac{1}{2}+\epsilon)^{t/2} (\frac{1}{2}-\epsilon)^{t/2} \\ &= \left(\frac{1-4\epsilon^2}{4}\right)^{t/2} \sum_{i=0}^{t/2} \binom{t}{i} \\ &\leq \left(\frac{1-4\epsilon^2}{4}\right)^{t/2} 2^t = (1-4\epsilon^2)^{t/2} \leq e^{-2\epsilon^2 t}, \end{aligned}$$

where the last inequality is again from the fact that for any x, $1 - x \leq e^{-x}$. ■

2.1 What Kind of Resource is Randomness?

One of the biggest open questions in complexity theory, is the question of whether randomness *really helps*, and what are the connections between the above classes of randomized algorithm. The following is one example of such a question that you should be able to answer after a bit of thinking.

Question 1 Are Monte Carlo algorithms at least as powerful than Las Vegas algorithms?

More formally, given a Las Vegas algorithm whose expected runtime is polynomial in the input length, can we turn it into a Monte Carlo Algorithm whose runtime is bounded by a polynomial in the input length?

Having figured out the answer to this question [think about it!], we might be tempted to ask:

Question 2 Are Las Vegas algorithms at least as powerful as Monte Carlo algorithms?

More formally, given a Monte Carlo Algorithm with polynomial runtime, can we turn it into a Las Vegas algorithm with polynomial expected runtime?

It turns out that this question is open! In complexity theory jargon, it's asking if $BPP = ZPP$ (for 2-sided error Monte Carlo algorithms) or if $RP = ZPP$ (for one-sided error Monte

Carlo algorithms). [But don't worry about this if you don't know the complexity theory jargon!]

We might also ask:

Question 3 Are deterministic algorithms as powerful as randomized algorithms?

More formally, given a randomized (Las Vegas or Monte Carlo) algorithm whose (expected) runtime is bounded by a polynomial in the input length, can we turn it into a deterministic algorithm whose runtime is bounded by a polynomial in the input length?

These questions are open too! In complexity theory jargon, this is asking if $P = BPP$ for 2-sided error Monte Carlo, or if $P = RP$ for one-sided error.

Surprisingly, while there are many instances of problems for which there seem to be much simpler and faster randomized algorithms, we do not have a rigorous proof of whether or not there problems for which randomness is essential for any efficient algorithm.

One way of approaching Question 3 is to ask whether we can simply replace the random coin flips via a sequence of deterministic bits that "look kind of random". For example, what if we run a randomized algorithm, but instead of flipping coins, we just use the bits of the binary representation of π? Surely that can't be too bad? We could also analyze how our analysis of randomization actually uses the randomness assumption, with an aim towards figuring out whether the algorithm would notice whether or not we fed it less-random bits.... These ideas fall under the broad umbrella of *derandomization*, and are discussed more in CS254.

There is also a large body of work understanding how to actually obtain random bits, and what can be done with imperfect randomness. For example, if the coins aren't quite unbiased or independent, are there ways of *distilling* or *extracting* the randomness? Given a large number of kindof-sortof-maybe random coins (that have some randomness, but maybe are correlated in unknown ways), can we produce a smaller number of almost perfectly random coins?

We won't discuss these topics much more in this course. Instead, our focus will be on designing and analyzing algorithms, assuming access to perfect random coins.

3 Randomized Polynomial Identity Testing

The first randomized algorithm that we will see is an algorithm for deciding whether a multivariate polynomial is actually 0: Given $P(x_1, x_2, \ldots, x_n)$, we want to know whether $P = 0$. This problem isn't as simple as it might seem at first. For example, consider

$$P(x_1, x_2, x_3) = -x_3^2(x_1 + x_2)(x_1 - x_2) + (x_1^2 + (1 + x_2)(1 - x_2))x_3^2 - x_3^2 - x_3^2(x_1 + x_2)(x_1 - x_2).$$

It turns out that, if you were to expand out this polynomial, all the terms cancel. A degree d polynomial in n variables could end up with $\gg \binom{n}{d} \approx n^d$ monomials—so the naive algorithm of simply expanding out all the monomials and then seeing what cancels, could, in the worst case, take time exponential in the length of the input representation of P.

Before describing an efficient randomized algorithm for this problem, it is worth pointing out that the task of identifying whether a polynomial is 0 or not is, in fact, an *incredibly* useful primitive, largely because polynomials occur everywhere. Here are two applications:

- Graph analysis: Graphs can be represented via adjacency matrices, and the determinants of these matrices (or variant of them) convey information about the structure of these graphs (especially the question of whether the determinant is 0 or not). Determinants are just polynomials! In particular, one can use this approach to efficiently test whether a graph has a perfect matching.
- Primality Checking: As we will see in a week or two, the polynomial $P(z) := (1+z)^n - 1 - z^n \mod n$ is 0 if, and only if, n is prime. Working modulo n is a twist that we won't discuss in this lecture, but we'll get to it later.

3.1 Schwartz-Zippel Polynomial Identity Test

This testing algorithm was published independently by Jack Schwartz and Richard Zippel in 1979 and should also be attributed to Richard DeMillo and Richard Lipton who published a version of it in 1978. Based on the fact that it was independently discovered at the same time by so many people, you might guess that it is fairly simple...

Algorithm 1
Polynomial Identity Test
Given a polynomial $P(x_1, \ldots, x_n)$:

1. Fix a finite set, S.
2. Choose n values, $r_1, \ldots, r_n$ independently and uniformly at random from S.
3. Evaluate $P(r_1, \ldots, r_n)$. If $P(r_1, \ldots, r_n) = 0$ output "$P = 0$" otherwise return "$P \neq 0$".

Theorem 3.1 *If $P = 0$, then the algorithm will always output "$P = 0$". If $P \neq 0$, then the algorithm will be correct with probability at least $1 - \frac{degree(P)}{|S|}$, where $degree(P)$ is the maximum sum of the degrees of terms in any monomial (e.g. $degree(x_1^3 x_2^2 + x_1^2 x_3^2) = 5$.)*

Proof: The first part of the proof is trivial, since if P is 0, then it will evaluate to 0. Hence, for the remainder, it suffices to consider the case that $P(x_1, \ldots, x_n)$ is a degree d polynomial. The proof leverages the following basic fact about univariate polynomials that we all learned at some point: a degree d polynomial in one variable can have at most d roots. Given this fact, we will prove the theorem by induction on the number of variables, n:

Base case, $n = 1$: The only way the algorithm will output the wrong answer is if $P(r_1) = 0$, in which case, by definition, r_1 is a root of P. Since P has degree d, by assumption, from

our fact about degree d univariate polynomials, $Pr[P(r_1) = 0] \leq \frac{d}{|S|}$, since at most d of the $|S|$ elements of our set can be roots of P.

Induction step: Assuming the theorem holds for polynomials in $\leq n-1$ variables, we will consider $P(x_1, \ldots, x_n)$, and try to reduce this case to an argument about the univariate polynomial $P(x_1, r_2, \ldots, r_n)$, obtained after plugging in the values for $x_2, \ldots, x_n$. To make this argument, we will leverage our induction hypothesis. Without loss of generality, assume that $P(x_1, \ldots, x_n) = x_1^k Q(x_2, \ldots, x_n) + T(x_1, \ldots, x_n)$ for some polynomials Q and T, such that $Q \neq 0$, and the maximum degree of x_1 in T is $< k$. Namely, we have factored out the monomials in P for which x_1 has the highest degree. [If x_1 does not actually appear in P, then we would do this argument for a variable x_i that does appear....since P is nonzero such a variable exists.]

Consider the event that $Q(r_2, \ldots, r_n) = 0$. By our inductive hypothesis, $Pr[Q(r_2, \ldots, r_n) = 0] \leq \frac{d-k}{|S|}$. If $Q(r_2, \ldots, r_n) \neq 0$, then what happens to P after we plug in $r_2, \ldots, r_n$? Well, then $P(x_1, r_2, \ldots, r_n)$ is a degree k nonzero polynomial of x_1. Applying our fact about univariate polynomials, the probability that this evaluates to 0 when we plug in r_1 for x_1 is at most $\frac{k}{|S|}$. Hence the probability the algorithm successfully evaluates to something other than 0 is at least

$$\left(1 - \frac{d-k}{|S|}\right)\left(1 - \frac{k}{|S|}\right) = 1 - \frac{d}{|S|} + \frac{(d-k)k}{|S|^2} \geq 1 - \frac{d}{|S|}.$$

■

3.2 Discussion

Given that the Schwartz-Zippel algorithm is the simplest thing imaginable, you might be tempted to wonder whether there is a simple deterministic algorithm that also works. Sadly, we do not know of any sub-exponential time deterministic algorithm! For certain special families of polynomials, deterministic algorithms are known (see e.g. the work of Ran Raz and Amir Shpilka [2]). In fact, Kabanets and Impagliazzo [1] have shown that if you find a deterministic polynomial time polynomial identity testing algorithm, this would imply some extremely strong complexity-theoretic results (such as $NEXP \not\subset polyCircuits$). We believe that these complexity theoretic results are true, and believe that a deterministic algorithm exists, but both seem far beyond what we can achieve with current approaches/techniques.

One reason to begin this class with polynomial identity testing is because the story is fairly faithful to our current understanding of randomized algorithms in general: a very simple/clean randomized algorithm exists, whose analysis is clever but fairly easy. We believe that an efficient deterministic algorithm exists, but that it is almost certainly extremely complicated.

References

[1] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004.

[2] Ran Raz and Amir Shpilka. Deterministic polynomial identity testing in non-commutative models. *Computational Complexity*, 14(1):1–19, 2005.