0368-4159: First Course in Derandomization 22/10/2018 – Lecture 2

Some complexity classes, Polynomial Identity Testing & Tail bounds

Lecturer: Amnon Ta-Shma
Scribe: Eyal Golombek

1 Complexity Classes Hierarchy

1.1 Motivation

We define some "standard" deterministic and randomized computational models. We also define a model for parallel computation, and discuss its connections with the "standard" complexity classes.

1.2 Notations and Basic Definitions

Definition 1 *(Deciding a Language) We say a Turing machine M decides a language L if M halts for every x and accepts iff $x \in L$.*

Definition 2 *(Time-Bounded Turing Machine) A Turing machine M is said to be time-bounded by some function $t : \mathcal{N} \to \mathcal{N}$ if the number of steps M makes on input x is bounded by $t(|x|)$.*

Definition 3 *(Space-Bounded Turing Machine) A Turing machine M is said to be space-bounded by some function $t : \mathcal{N} \to \mathcal{N}$ if the size of the **work-tape** that M uses while working on input x is at most $t(|x|)$. $Space(s(n))$ denotes the set of languages decidable by a Turing machine running in space $s(n)$.*

Note that the bound is only on the work-tape. This allows machines to be space bounded with sub-linear functions. This means the machine can read the entire input, but can't "remember" all of it at once.

Definition 4 *(Randomized Computation) Probabilistic Turing machines have an Read-once tape, initialized with uniformly chosen bits. We define three types of randomized classes:*

- ***R Time/Space**: One-sided error. For every input not in the language, the acceptance probability is zero, for every input in the language, the acceptance probability (over the choice of the random tape) is at most half.*
- ***B Time/Space**: Bi-sided error. Similar to R Time/Space, except we allow the machine to err with bounded probability in both directions.*
- ***Z Time/Space**: Zero sided error. Similar to the above, except we don't allow the machine to err. Instead, the machine is allowed to output $\perp$ which means "I Don't Know", and for every input, the probability of $\perp$ is at most half.*

1.3 Simple containments

$$\mathsf{DSPACE}(Log(n)) \subseteq \mathsf{DTIME}(Poly(n)) \subseteq \mathsf{DSPACE}(Poly(n)) \subseteq \mathsf{DTIME}(Exp(n))$$

1.4 Boolean Circuits

Another way to model computation, instead of using Turing machines, is Boolean Circuits.

Definition 5 *(Boolean Circuits) A Boolean Circuit is a directed a-cyclic graph (V, E), with a subset $I \subset V$ of input nodes and $O \subset V$ output nodes. Every input node has in-degree zero, and every output node has out-degree zero. Every other node is assigned a Boolean function out of $\{And, Or, Not\}$. The* size *of a Boolean Circuit is defined to be the number of edges +the number of vertices in the circuit. The* depth *of the circuit is the length of the longest path from an input node to an output node in the circuit.*

Every Boolean circuit has some hard-coded input size. Usually we want to discuss languages that have inputs of varying size.

Definition 6 *(Family of Circuits) A Family of circuits $\{C_n\}_{n\in N}$ is a set of circuits, such that C_n has input of size n. We say the circuit family has size bounded by some function $S : \mathcal{N} \to \mathcal{N}$ if $S(n)$ bounds the size of C_n for every n. Same for family depth. We say a family of circuits $\{C_n\}_{n\in N}$ decides some language L if for every x, $C_{|x|}(x) = 1$ iff $x \in L$*

A computation is said to be **Non-Uniform** if for every input length we allow a different algorithm to operate. (The different algorithms don't necessarily have some connection to each other). Every language $L \subset \{0,1\}^*$ has a non-uniform family of circuits $\{C_n\}_{n\in N}$ with size bounded by $O(n \cdot 2^n)$ that decides L, no matter the complexity of L. Each circuit C_n will simply have a hard-coded table with 2^n entries, every entry of the table corresponds to a n bit string, and the table would indicate if we need to accept or reject.

It is possible that every language in NP has a polynomial size, non-uniform family of circuits that solves it. This does not imply that $\mathcal{P} = \mathsf{NP}$. However if we were to find such family that solves some $\mathsf{NP} - Complete$ problem, for instance $3SAT$, it would collapse the polynomial hierarchy into $\mathsf{PH} = \Sigma_2$

Definition 7 *(Uniform Circuit Families) We say a Circuit family $\{C_n\}_{n\in N}$ is log-uniform if there exists some Turing machine M that on input 1^n outputs the description of C_n in $O(logn)$ space.*

Intuitively speaking it means there is some pattern that generates all the circuits in the family, and this pattern can be generated efficiently.

1.5 Parallel Computation as Circuit Family

As we've seen before, the real-world RAM model of computation can be transformed into a theoretical computational model using circuit families. We would like to do the same to the Parallel-RAM model. Intuitively speaking, when thinking of a Boolean circuit, we can divide all nodes into layers, by their distance from an input node. We can think of each node as a distinct processor, and each layer as one step of computation. Every layer is performed sequentially after the previous one, and all computations in the same layer are done in parallel. Thus every processor can read the outputs of processors from previous layers. Following this intuition, efficiency will require a polynomial number of processors, which translates to polynomial sized circuits, and will require poly logarithmic number of computation steps, which translates to poly-log depth.

Definition 8 *(NC) (Uniform / non-uniform) NC^k is the class of all languages L that have a (uniform / non-uniform)family of circuits $\{C_n\}_{n\in N}$ with fan-in of 2, that decides L and where $|C_n| < poly(n)$ and $depth(C_n) < O(log^k n)$*

Unless we say otherwise, we work with the *uniform* classes.

We let $\mathsf{NC} = \cup_k \mathsf{NC}^k$. We define AC^k to be as NC^k, except that we allow an arbitrary fan-in for the Boolean gates. Similarly, $\mathsf{AC} = \cup_k \mathsf{AC}^k$. We have:

- $\mathsf{NC}^0 \subset \mathsf{AC}^0 \subset \mathsf{NC}^1 \subset \mathsf{AC}^1$...
- $\mathsf{NC} = \mathsf{AC}$.
- $\mathsf{NC} \subseteq \mathcal{P}$, since the generation of the circuit is $Space(logn)$ which is polynomial time. After we have generated the circuit we can simply emulate it in polynomial time.
- $\mathsf{NC}^k \subseteq \mathsf{DSPACE}(O(log^k n))$
- $\mathsf{RSPACE}(logn) \subset \mathsf{NSPACE}(logn) \subset \mathsf{DSPACE}(Log^2(n))$
- $\mathsf{RSPACE}(logn) \subset \mathsf{BPSPACE}(log^n) \subset \mathsf{AC}^1 \subset \mathsf{NC} \subset \mathcal{P}$
- $\mathsf{NC} \subset R\mathsf{NC} \subset \mathsf{RP} \subset \mathsf{NP}$
- $\mathcal{P} \subset \mathsf{RP} \subset \mathsf{BPP}$
- $\mathsf{AC}^0 \neq \mathsf{NC}^1$. The parity of n bits is known to be in NC^1 and known to **not** be in AC^0.

1.5.1 Examples

- **ADD**: Addition of two n bit numbers. Intuitively every output bit depends on more than a constant number of input bits, so $ADD \notin \mathsf{NC}^0$, but we can prove $ADD \in \mathsf{AC}^0$.
- **Mul**: Multiplication of two n bit numbers: $Mul \in \mathsf{NC}^1$.
- **Matrix Multiplication** $\in \mathsf{NC}^1$. We have seen an algorithm in the previous lesson.
- **Determinant Calculation** $\in \mathsf{AC}^1$.

1.6 Open Questions

- $\mathsf{BPL} = \text{Ł}$?
- $R\mathsf{NC} = \mathsf{NC}$?
- $\mathcal{P} = \mathsf{BPP}$?

2 Polynomial Identity Testing problem (PIT)

2.1 Intuition

We look for a canonical example of a language that we know to be in RP, but is not known to be in $\mathcal{P}$. So far we've seen the perfect-matching language (PM) that is in $R\mathsf{NC} \subseteq \mathsf{RP}$, but is not known (yet) to be in NC (it is known to be in "quasi"-NC). However, PM is known to be in $\mathcal{P}$ using flow algorithms. In this section we define PIT. We show it is in coRP. PIT is not known (yet?) to be in $\mathcal{P}$.

Definition 9 *An **Arithmetic Circuit** over some field $\mathcal{F}$ is defined similarly to Boolean circuits (A directed graph with input/output nodes), except that the operations each node can perform are no longer Boolean And/Or/Not, but rather are Addition/Multiplication of elements in the field $\mathcal{F}$.*

Remark 10 *Any arithmetic circuit defines some syntactic polynomial $p(x_1, ..., x_n)$, where $x_1, ..., x_n$ are the input nodes. If the circuit uses only the constants 0 and 1, this polynomial does not depend on the specific field. It's simply the formula which defines what additions / multiplications this circuit performs.*

Definition 11 *(The language **PIT**) The input is a description of an arithmetic circuit C. Let $p(x_1, ..., x_n)$ be the polynomial describing the operation of C. The input C is in the language PIT if $p \equiv 0$. (Equality is over $\mathbb{C}[X]$, where $\mathbb{C}$ is the complex field. This means that p is identical to the zero polynomial.)*

Remark 12 *(**Reduction from** PM **to** PIT) Recall the algorithm we described in the previous class for deciding the PM language. The final step of the algorithm is to decide whether the polynomial describing the determinant is zero or not. This means that any algorithm for solving PIT can be used to solve PM.*

There is a simple randomized algorithm for PIT: Sample random inputs to the circuit and test if the calculation of the polynomial on those inputs is equal to zero. By utilizing the Schwartz-Zippel Lemma, we hope to get some good bound on our error probability. We start with a Naive algorithm:

Given as input an arithmetic circuit $C(x_1, ..., x_n)$ of size $|S| \leqslant p(n)$ and degree d:

1. Choose $A \subseteq R$ of size $|A| = 2d$ (Can be the first integers, for instance)
2. Choose $a_1, ..., a_n \in A$ randomly uniformly and independently.
3. Calculate $p(a_1, ..., a_n)$. If it's zero accept, otherwise reject.

Clearly, if p is the zero polynomial, we always get 0 and accept, whereas if p is not the zero polynomial, by the Schwartz-Zippel theorem we reject with probability at least $1 - \frac{d}{|A|} \geq \frac{1}{2}$. Also, w.l.o.g., we can take $d = 2^S$ (or $d = 2^{2S}$ if we want lower error) as:

Claim 13 *Let $C(x_1, ..., x_n)$ an arithmetic circuit of size S. Let $p(x_1, ..., x_n)$ the polynomial representing C. Then $deg(p) \leqslant 2^S$.*

Proof This is obvious since every gate can increase the degree by up to a factor of 2.

What remains is to check that the algorithm runs in polynomial time (alas, it does not).

- We need to sample n elements out of A. The size of A is 2^{2S} thus making every sample require $2S$ random bits. That's fine! (No real issue here)
- We need to calculate $p(a_1, ..., a_n)$. Recall that we chose $a_i \in A$ and the size of A is 2^{2S}. If p is for instance x^{2^S} then calculating p on inputs of size 2^S (or even just 2) will explode. Specifically, this results in values of size $2^{S2^S} \approx 2^{2^S}$. Those kind of values take at least $O(2^S)$ bits to represent, i.e., exponential space.

To overcome this we extend the Naive algorithm by choosing randomly some prime $2^{2S} \leqslant q \leqslant 2^{4S}$. From here on, all calculations will be done modulo q. Continue operating as the Naive algorithm.

- If p is indeed the zero polynomial, it is still zero after performing modulo, no matter the prime.
- Selecting a random prime can be done efficiently by randomly sampling a number and performing primality testing in polynomial time. The density of primes gives us very high rates of success of finding a prime in polynomial time (do it as an exercise!).

To prove correctness, do Schwartz-Zippel over the field with p elements. This is left as an exercise (in HW1).

PIT belongs to the family of languages that give you a code (an a-cyclic arithmetic circuit in our case) and asks you to understand what the code does (does it compute the zero polynomial?). Offhand, it seems PIT is a difficult problem for the deterministic model. Later on in the class, we will have an opportunity to reconsider our opinion.

3 Tail Bounds

A natural question that arises when working with random variables is: *"How probable is it for a random variable to deviate from its expected value?"*. We will now show 3 widely used tail bounds that help us answer this question.

Our settings: Let $X_1, ..., X_n$ be Boolean random variables ($X_i \in \{0,1\}$) such that $\Pr[X_i = 1] = \mathbb{E}[X_i] = \mu_i$. We define $X = \sum_{i=1}^{n} X_i$ making $\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \mu_i = \mu$

The questions we would like to answer are:

- $\Pr[X > (1+\delta)\mu] <?$
- $\Pr[X < (1-\delta)\mu] <?$
- $\Pr[|X - \mu| > \delta\mu] <?$

3.1 Markov's Inequality

Theorem 14 *Let X be as defined above, and assume $X \geqslant 0$ then:*

$$\Pr[X \geqslant A] \leqslant \frac{E[X]}{A}.$$

Proof *Intuitively speaking, if it were true that for more than $\frac{\mathbb{E}[X]}{A}$ fraction of the samples X gets a value larger than A, the expected value of X had to be higher. Formally:*

$$\begin{aligned} \mathbb{E}[X] &= \sum_{x \in X} x \Pr[X = x] = \sum_{x < A} x \Pr[X = x] + \sum_{x \geqslant A} x \Pr[X = x] \\ &\geq \sum_{x \geq A} x \Pr[X = x] \geq A \sum_{x \geqslant A} \Pr[X = x] = A \Pr[X \geqslant A]. \end{aligned}$$

Using the fact that X is always positive.

For arbitrary random variables the bound can be quite tight. For example, assume $X_1 = X_2 = ... = X_n$ are random bits. Clearly: $\mathbb{E}[X_i] = 1/2$, $\mathbb{E}[X] = n/2$. Because all random variables take the same value, there are only two possible outcomes, either all bits are 1 or all bits are 0, and both outcomes have probability $1/2$. Thus, $\Pr[X \geqslant \frac{2}{3}n] = \Pr[X = n] = 1/2$, whereas Markov gives $\frac{n/2}{2n/3} = \frac{3}{4}$.

3.2 Chebyshev's inequality

Theorem 15 *Given X as defined above, (but no need for positivity of X):*

$$\Pr[|X - \mu| \geqslant A] \leqslant \frac{Var[X]}{A^2}.$$

Proof *Using Markov's inequality on $|X - \mu|$, and using the definition of variance:*

$$\Pr[|X - \mu| \geqslant A] = \Pr[(X - \mu)^2 \geqslant A^2] \leqslant \frac{E[(X - \mu)^2]}{A^2} = \frac{Var[X]}{A^2}.$$

3.3 k-wise Independence, and tail bounds for k-wise independent distributions

Definition 16 *We say random variables $X_1, ..., X_n$ are K-wise independent if every subset of them of size K are independent. I.e., for every $\Lambda \subset [n]$ of size K, and every $a_1, \ldots, a_k$:*

$$\Pr[\bigwedge_{i\in\Lambda} X_i = a_i] = \Pi_{i\in\Lambda} \Pr[X_i = a_i].$$

We reiterate the special case of $k = 2$:

Definition 17 *$X_1, ..., X_n$ are pair-wise independent if for every pair X_i, X_j where $i \neq j$, X_i and X_j are independent.*

Example 18 *Let X_1, X_2, X_3 be uniformly distributed over all $(a, b, c) \in \{0, 1\}^3$ such that $a+b+c = 0 \bmod 2$. Verify that X_1, X_2, X_3 are pair-wise independent. Clearly, X_1, X_2, X_3 are not independent.*

The reason we want k-wise independence is that independence gives us more power in creating tail bounds (As we will see in the following section). The reason we use K-wise independence instead of complete independence is that it requires much less random bits to generate (we will see that later on).

Theorem 19 *Suppose $X_1, ..., X_n$ are identically distributed boolean random variables, **pair-wise independent** and $\Pr[X_i = 1] = p$. As before $X = \sum_{i=1}^n X_i$. Then:*

$$\Pr[|X - \mu| \geqslant A] \leqslant \frac{np(1-p)}{A^2}$$

Proof *Theorem is using Chebyshev's inequality, and noticing that:*

$$Var(X) = \mathbb{E}[(X-\mu)^2] = \mathbb{E}[(\sum_{i=1}^n X_i - \mu)^2].$$

Denote $Z_i = X_i - \mu_i$, clearly $\mathbb{E}[Z_i] = 0$ and

$$Var(X) = \mathbb{E}[(\sum_{i=1}^n Z_i)^2] = \mathbb{E}[\sum_{i,j} Z_i Z_j] = \sum_{i,j} \mathbb{E}[Z_i Z_j] = \sum_{i=1}^n \mathbb{E}[Z_i^2] + \sum_{i\neq j} \mathbb{E}[Z_i Z_j].$$

Using the pair-wise independence we get:

$$Var(X) = \sum_{i=1}^n \mathbb{E}[Z_i^2] + \sum_{i<j} \mathbb{E}[Z_i]\,\mathbb{E}[Z_j] = \sum_{i=1}^n \mathbb{E}[Z_i^2] = \sum_{i=1}^n Var[X_i] = np(1-p).$$

The final equality is due to the fact that $Var[X_i] = p(1-p)$. Using Chebyshev's inequality, we get the result.

Choosing $A = \alpha pn$ we get:

$$\Pr[|X - \mu| \geq \alpha pn] \leqslant \frac{np(1-p)}{\alpha^2 p^2 n^2} = O(\frac{1}{n}).$$

More generally, for any $2k$ - wise independent set of variables in this settings, by bounding the $2k$ moment, as seen above, will produce an asymptotic bound of $O(\frac{1}{n^k})$. We demonstrate it below for $k = 4$ (and a special setting for the random variables).

Suppose $X_1, ..., X_n$ are identically distributed random variables over $\{-1, 1\}$ with $\Pr[X_i = 1] = \Pr[X_i = -1] = 1/2$. Further assume $X_1, \ldots, X_n$ are 4-wise independent. Then:

$$\Pr[|X - \mu| \geqslant \alpha pn] \leqslant O(\frac{1}{n^2}).$$

Proof *It's clear that $\mathbb{E}[X_i] = \mu_i = 0$, thus making $Z_i = X_i - \mu_i = X_i$. Recall that $X_i \in \{-1, 1\}$ thus making $X_i^3 = X_i$ and $X_i^4 = X_i^2 = 1$. Using Chebyshev's inequality on the 4th moment:*

$$\Pr[|X - \mu| \geqslant A] = \Pr[(X - \mu)^4 \geqslant A^4] \leqslant \frac{\mathbb{E}[(X - \mu)^4]}{A^4}.$$

Bounding the 4th moment we get:

$$\mathbb{E}[(X - \mu)^4] = \mathbb{E}[(\sum_{i=1}^{n} X_i)^4] = \mathbb{E}[\sum_{i,j,k,l} X_i X_j X_k X_l] = \sum_{i,j,k,l} \mathbb{E}[X_i X_j X_k X_l]$$

Recall that $\mathbb{E}[X_i] = 0$ for every i. Using the 4-wise independence we know that the expectancy of multiplication of different random variables is equal to the multiplication of their expectancies. Thus the only elements in this sum that don't zero out are those where all 4 of the variables are the same one, or if they appear in two pairs.

$$\mathbb{E}[(X - \mu)^4] = \sum_i \mathbb{E}[X_i^4] + \sum_{i,j} \mathbb{E}[X_i^2]\,\mathbb{E}[X_j^2].$$

Recall that $X_i^4 = X_i^2 = 1$. Thus,

$$\mathbb{E}[(X - \mu)^4] = n + \binom{n}{2}\binom{4}{2} = O(n^2).$$

Finally using the bound on the 4th moment, we get:

$$\Pr[|X - \mu| \geqslant \alpha pn] \leqslant \frac{\mathbb{E}[(X - \mu)^4]}{\alpha^4 p^4 n^4} = O(\frac{1}{n^2}).$$