University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Glenn Sun
Date: November 12, 2020

LECTURE
13

The Sipser–Gács Theorem and Other Topics in Randomness

In this lecture, we continue our discussion of randomized complexity classes by proving the Sipser–Gács theorem. We also introduce the notion of a zero-error randomized algorithm, understand how it relates to other randomized complexity classes, and wrap up our discussion of randomness by discussing biased coin flips.

13.1 The Sipser–Gács theorem

As alluded to in the previous lecture, we now present a proof that BPP is contained in the second level of the polynomial hierarchy. This is quite surprising and non-trivial, since it is not very clear why randomness should be related at all to the polynomial hierarchy. Although Sipser and Gács first proved these results, the proof presented here is due to Lautemann [1].

THEOREM 13.1 (Sipser–Gács theorem). $\text{BPP} \subset \Sigma_2 \cap \Pi_2$.

Proof. It suffices to show that $\text{BPP} \subset \Sigma_2$. Then the fact that $\text{BPP} \subset \Pi_2$ comes from the fact that $\text{co}\,\text{BPP} = \text{BPP}$ and $\text{co}\,\Sigma_2 = \Pi_2$.

We will demonstrate that $\text{BPP} \subset \Sigma_2$ by simply writing a formula. Recall that a language L is in Σ_2 if there exists a polynomial time Turing machine such that:

$$x \in L \iff \exists u_1 \, \forall u_2 \, M(x, u_1, u_2) = 1.$$

Let $L \in \text{BPP}$, so that without loss of generality we may a fix a polynomial time Turing machine M that computes L with $1 - \frac{1}{2^n}$ probability of success. We claim that

$$x \in L \iff \exists u_1, \ldots, u_{\text{poly}\, n} \, \forall r \, \vee_i \, M(x, r \oplus u_i) = 1.$$

In other words, $x \in L$ if and only if for the particular machine that we fixed, there exists a polynomially large set of strings $\{u_1, \ldots, u_{\text{poly}\, n}\}$ such that translating any random string r by one of the u_i forces M to accept. Note that this is not obvious at all because there

are $2^{\text{poly}(n)}$ different random strings and M rejects on a $\frac{1}{2^n}$-fraction of them, which is still exponentially large.

If we show the claim, it immediately implies that BPP $\subset \Sigma_2$ because we may consider the strings $u_1, \ldots, u_{\text{poly}\, n}$ to be concatenated as a single string u, and we may choose a Turing machine $M'(x, u, r)$ that computes the value of $\vee_i M(x, r \oplus u_i)$ by simulating M. Simulating M takes polynomial time, and there are polynomially many u_i, so this is polynomial-time simulation.

To show the claim, let us first parse the statement. Let A_x denote the set of random strings on which M accepts x, i.e. $A_x = \{r : M(x, r) = 1\}$. Suppose that $u_1, \ldots, u_{\text{poly}\, n}$ have already been selected. Again, the expression $\forall r \vee_i M(x, r \oplus u_i) = 1$ means that translating any random string by one of u_i forces M to accept. Equivalently, we may undo some translation from some accepting string to recover any random string. Since undoing $\oplus$ (xor) is the same as doing it again, we get that $\forall r \vee_i M(x, r \oplus u_i) = 1$ if and only if $\bigcup_i (A_x \oplus u_i)$ forms the entire space of random strings $\{0,1\}^{\text{poly}\, n}$.

The main idea is to show that for $x \notin L$, the sets $A_x \oplus u_i$ are too small and too few to cover the whole space, whereas for $x \in L$, the sets $A_x \oplus u_i$ are so large and numerous that most choices of u_i end up covering the whole space. In a picture,

Now, consider $x \notin L$, and we wish to show that for any choice of $\{u_1, \ldots, u_{\text{poly}\, n}\}$, we have $\bigcup_i (A_x \oplus u_i)$ misses some random string. This is the easy direction. In particular, because A_x is at most a $\frac{1}{2^n}$ fraction of all random strings and the number of different translations is polynomial, union bound implies that some random string is not covered.

For the harder direction, consider $x \in L$, and we wish to pick $u_1, \ldots, u_{\text{poly}\, n}$ such that $\bigcup_i (A_x \oplus u_i) = \{0,1\}^{\text{poly}\, n}$. The technique we will use is called the *probabilistic method*. In particular, we will pick $u_1, \ldots, u_{\text{poly}\, n}$ uniformly at random, and show that the probability of the bad event is less than 1. (Keep in mind that here, r is not being sampled randomly, so probabilities are only over $u_1, \ldots, u_{\text{poly}\, n}$.) The bad event is when there exists a random string r such that $r \notin \bigcup_i (A_x \oplus u_i)$. Note the following two facts:

1. For any fixed r, the events $r \notin A_x \oplus u_i$ (for each u_i) are independent, since the u_i are chosen independently.

2. The probability that $r \notin A_x \oplus u_i$ is at most $\frac{1}{2^n}$ because A_x contains at least a $(1 - \frac{1}{2^n})$-fraction of all appropriate-length strings.

Then, we can compute

$$\begin{aligned}
\mathbb{P}(\exists r : r \notin \textstyle\bigcup_i (A_x \oplus u_i)) &\le \sum_r \mathbb{P}(r \notin \textstyle\bigcup_i (A_x \oplus u_i)) && \text{(union bound)} \\
&= \sum_r \prod_{u_i} \mathbb{P}(r \notin A_x \oplus u_i) && \text{(by 1)} \\
&\le \sum_r \prod_{u_i} \frac{1}{2^n} && \text{(by 2)} \\
&= 2^{\operatorname{poly} n} \left(\frac{1}{2^n}\right)^{\operatorname{poly} n}
\end{aligned}$$

If we choose the number of u_i to be a polynomial greater than the number of random bits used, then this number is clearly less than 1. Hence by the probabilistic method, there exists $u_1, \ldots, u_{\operatorname{poly} n}$ such that the claim is true, which completes the proof. □

Although the Sipser–Gács theorem was a major breakthrough in computer science, many people actually believe a stronger statement that BPP is contained as low as P, i.e. the zeroth level of the polynomial hierarchy. However, this is the frontier of current knowledge about randomized complexity classes. We also want to highlight here the non-trivial application of the probabilistic method, without which it would be been prohibitively difficult to identify which random strings $u_1, \ldots, u_{\operatorname{poly} n}$ to take.

13.2 Zero-error randomized algorithms

So far, we have used randomness at the cost of accuracy, but in fact this need not be the case. Instead, it is often useful to guarantee complete accuracy, and have the running time be randomized instead. Recall that quicksort is one basic example of a zero-error randomized algorithm, which runs in $O(n \log n)$ time in expectation, but may take as long as $O(n^2)$ time. Although the running time of quicksort is bounded by $O(n^2)$ on every random string, we don't require such a bound for zero-error randomized algorithms in general, allowing the machine to take arbitrarily long time on a small fraction of random strings, as long as the expected running time remains polynomial.

Definition 13.2 (ZPP). A language L is in ZPP if there exists a Turing machine M such that M runs in expected polynomial time and computes L correctly whenever it halts.

Surprisingly, we can draw an exact equality between ZPP and some classes that we already know! This is the main result of this section, presented below.

Proposition 13.3. $\mathrm{ZPP} = \mathrm{RP} \cap \mathrm{co\,RP}$.

Proof. ($\subset$) Suppose that L admits a ZPP algorithm that runs in $T(n)$ time, where $T(n)$ is $\operatorname{poly} n$. Because polynomials are time-computable functions, we may simply run the ZPP algorithm for $3T(n)$ steps and check to see if it has halted yet. If it has halted, we return its output, and if it has not halted, we return a default value.

To show that $\mathrm{ZPP} \subset \mathrm{RP}$, we would set the default value to 0, and for $\mathrm{ZPP} \subset \mathrm{co\,RP}$, would set it to 1. This is because a default value of 0 guarantees that every no-instance is computed correctly, and likewise a default value of 1 guarantees yes-instances. Either way,

the probability of being wrong for the remaining instances is bounded by the probability of landing in the default case, which happens with probability at most $\frac{1}{3}$ by Markov's inequality, which shows $\mathrm{ZPP} \subset \mathrm{RP} \cap \mathrm{co\,RP}$.

($\supset$) If L admits both RP and co RP algorithms, note that the RP algorithm makes no mistakes on no-instances and the co RP algorithm makes no mistakes on yes-instances. Hence, we may run both in a loop, return if they agree on the output, else retrying with a new random string if they disagree. Clearly this is always correct.

To analyze the running time, simply note that the expected number of iterations is constant. If we take the RP and co RP algorithms to have probability of being wrong at most $\frac{1}{3}$, note that the probability that they disagree is exactly the probability that one of them is wrong, so the probability of needing to retry is at most $\frac{2}{3}$ by union bound. This means the number of iterations is bounded by a geometric random variable, so the expected number of iterations is constant. Each iteration takes polynomial time, so the algorithm runs in expected polynomial time. □

Our picture of randomized complexity classes is now as follows:

13.3 Simulating coin tosses

One technicality that we've avoided mentioning so far is how to actually choose random elements from sets. We are given a string of random bits, so it is easy to choose random elements from sets of the size 2^k for any k by simply reading k random bits and interpreting it as a binary number. However, it is less obvious how to, for example, choose a random element from the set $\{1, 2, 3\}$, or how to flip a coin that has a 70% chance of landing heads. We will just discuss the coin flipping problem, since one can clearly choose a random element from a set by flipping sufficiently many biased coins.

Intuitively, if we wanted to simulate a coin that outputs 1 with probability $\frac{1}{3}$, we could flip a fair coin twice, denote 3 of the 4 outcomes as terminating cases (of which one case outputs 1 and the other two output 0), and in the last case repeat the process. But this strategy doesn't work for irrational probabilities like $\frac{1}{\sqrt{2}}$ or $\frac{1}{\pi}$. The fact that the process described above repeats can be likened to the fact that a number is rational if and only if its binary representation (equivalently, any integer base at least 2) either repeats or terminates. Hence, to generalize the basic strategy to allow irrational probabilities, we can use the binary representation directly.

PROPOSITION 13.4. *Given a fair coin and a number $0 < p < 1$ such that the nth bit of the binary expansion of p can be computed in $\operatorname{poly} n$ time, one can simulate a p-biased coin with $O(1)$ expected time.*

Proof. Denote by $0.p_1p_2...$ the binary expansion of p. For $i = 1, 2, \ldots$, simply take a random bit r_i and check if $r_i = p_i$. If they are equal, then output their value (whether 0 or 1), and otherwise continue to the next iteration.

To analyze the correctness, note that in each iteration, there is $\frac{1}{2}$ chance of terminating because one may understand p_i to be predetermined and r_i to be taken from a fair coin flip. Hence the chance of terminating in the ith iteration is exactly $\frac{1}{2^i}$. Given the algorithm terminates in iteration i, the probability of outputting 1 is exactly p_i (either 0 or 1). Hence, the probability of outputting 1 overall is $\sum_{i=1}^{\infty} \frac{1}{2^i} p_i$, which is exactly the definition of the binary expansion of p.

To analyze the running time, note that if the algorithm terminates in iteration i, then the algorithm was required to calculate $p_1, \ldots, p_i$, which takes $\operatorname{poly} i$ time. The expected running time is thus $\sum_{i=1}^{\infty} \frac{1}{2^i} \operatorname{poly} i$. No matter the polynomial, this infinite series converges to a fixed constant by any number of common tests from analysis, i.e. the ratio test. In fact, this suggests that we may allow computing the binary expansion of p to take up to $2^n/n^{1+\epsilon}$ time! □

Even if we still require that p be computable in polynomial time, it turns out that this restriction permits almost all numbers that we care about. All algebraic numbers like $\frac{1}{3}$ or $\frac{1}{\sqrt{2}}$ can be computed efficiently by simply running a root-finding algorithm on the appropriate polynomial. And most transcendental numbers that we care about, such as $\frac{1}{e}$ and $\frac{1}{\pi}$, also admit polynomial time computation algorithms. However, it is true that only countably many numbers can be computed at all since there are only countably many Turing machines, so almost all real numbers are not allowed.

It is also interesting to prove a converse statement about simulating fair coins with biased coins. In fact, it's possible to simulate fair coins without even knowing the probabilities of the biased coin! The key idea is to find two equally likely events no matter what the bias of the coin is. Also, since we are given a biased coin and asked to simulate a fair coin, it no longer matters whether or not p is computable in polynomial time.

PROPOSITION 13.5. *Given a p-biased coin (even if the value of p is not given), one can simulate a fair coin using $O(\frac{1}{p(1-p)})$ expected time.*

Proof. Notice that no matter what p is, if we flip the coin twice and let the results be denoted a and b, then $\mathbb{P}(a = 0 \wedge b = 1) = \mathbb{P}(a = 1 \wedge b = 0) = p(1-p)$. Hence, we may simply try this, output 0 in the first case and 1 in the second case, and if neither case happens, try again until one of the two happens. Clearly, this simulates a fair coin, and the number of iterations forms a geometric distribution with ratio $p(1-p)$, so the expected time is $O(\frac{1}{p(1-p)})$ as desired. □

As an aside, these are both ZPP algorithms (if p is considered a constant), so both can be used in either RP or co RP algorithms. By running them for a sufficiently long time, we can reduce the probability of error to be minuscule (at most $\frac{1}{\operatorname{poly} n}$), even after taking union bound for up to $\operatorname{poly} n$ coin tosses. Hence, they are perfectly safe to use as subroutines inside algorithms designed to be RP or co RP.

This concludes our initial discussion of randomness, although we'll see many applications of randomness and other complexity classes involving randomness in the future. In the next lecture, we will shift gears and discuss a new model of computation not based on Turing machines at all — Boolean circuits.

References

[1] C. Lautemann. Bpp and the polynomial hierarchy. *Information Processing Letters*, 17(4):215 – 217, 1983.